5 Métodos de Ordenamiento

- 5.1 Métodos Internos de Ordenamiento
  - 5.1.1 Burbuja
  - 5.1.2 Quicksort
  - 5.1.3 Heapsort
  - 5.1.4 Inserción Simple
  - 5.1.5 Shellsort
- 5.2 Métodos Externos de Ordenamiento
  - 5.2.1 Mezclas
  - 5.2.2 Distribución Simple
  - 5.2.3 Radix

5 Métodos de Ordenamiento

Métodos de Ordenamiento

- Internos.- El espacio extra que se requiere para realizar el ordenamiento, es tan pequeño que se considera nulo.
- Externos.- Se requiere una cantidad de espacio extra proporcional al número de datos a ordenar.

Registro.
Cada uno de los elementos que conforman un archivo

97040225	3608	75	EXT
97040225	3609	78	NOR
99040698	3609	85	NOR
99040698	3610	86	REG1
97040225	3611	80	NOR
97040225	3622	75	REG2
97040225	5503	70	EXT
98040150	5503	93	NOR
99040698	5503	81	EXT
98040150	5508	100	NOR
98040150	5566	92	NOR
98040150	5577	75	NOR
98040150	5578	80	REG1
99040698	6615	83	ESP

Llave.
Campo o conjunto de campos que determinará el orden del archivo.

Archivo.
Nos podremos referir a aquellos archivos en Disco Duro o SSD o a una colección en memoria RAM (una matriz por ejemplo).

5.1 Ordenamiento Interno

Convenciones

Para simplificar el análisis de los diversos métodos que estudiaremos

- Los archivos de ejemplo se compondrán únicamente de llaves aunque los registros de las aplicaciones prácticas casi siempre contienen otros datos aparte de la llave.
- Las llaves aparecerán como valores numéricos.
- Las llaves solo se compondrán de un dato (en los casos prácticos puede ser cualquier número de datos).
- Se describirán los métodos de ordenamiento asumiendo que se desea organizar el archivo en forma ascendente (para hacerlo a la inversa los cambios son pequeños).
- En los casos prácticos los datos que contiene cada registro (aparte de la llave) son tan importantes como la llave misma y debemos tener en cuenta que cuando hablemos de **mover la llave** de un registro a otro lugar del archivo, realmente nos referiremos a **mover el registro completo**.

97040225	3608	75	EXT
97040225	3609	78	NOR
99040698	3609	85	NOR
99040698	3610	86	REG1
97040225	3611	80	NOR
97040225	3622	75	REG2
97040225	5503	70	EXT
98040150	5503	93	NOR
99040698	5503	81	EXT
98040150	5508	100	NOR
98040150	5566	92	NOR
98040150	5577	75	NOR
98040150	5578	80	REG1
99040698	6615	83	ESP

5.1.1 Método de la Burbuja

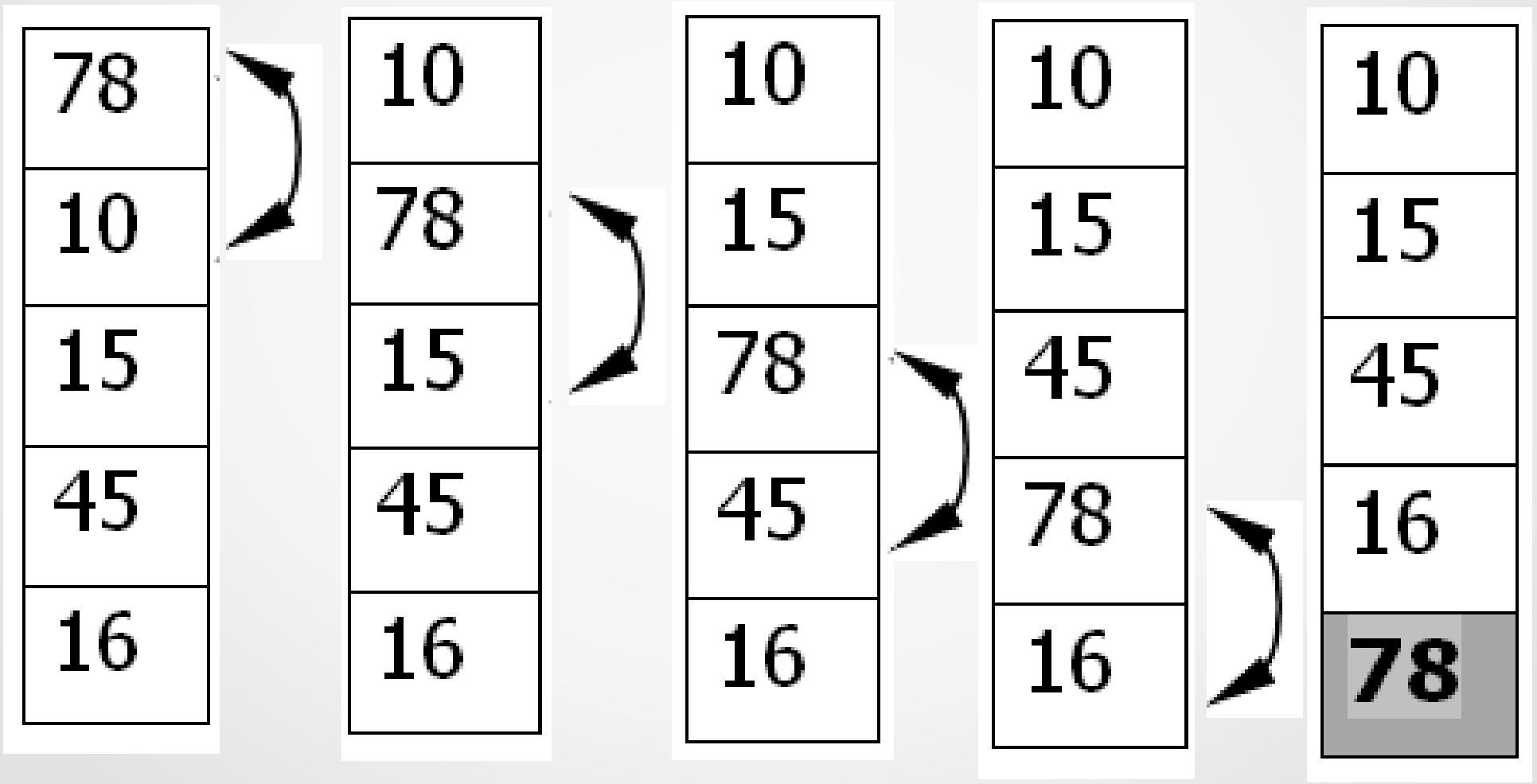
A. Logra el objetivo realizando varios **"pasos"** al archivo comparando pares de llaves en posiciones adyacentes.
B. Si la **comparación** determina que están fuera de orden, **intercambiarán sus lugares**.
C. Se inicia comparando la **1ª y 2ª llaves**.
D. Luego se compararán la **2ª y 3ª llaves** (la segunda puede ser la que antes era la primera).
E. Y así sucesivamente hasta comparar **la penúltima** con la **última** llaves.

- Los pasos anteriores logran que el **valor más grande** sea precipitado al fondo del archivo.
- Si en esta etapa, **no se realizan intercambios**, significa que el archivo está ordenado y el proceso debe terminar.
- En caso de que se presente **al menos un intercambio**, el razonamiento obliga a repetir los pasos arriba descritos.
  - Iniciando con las **llaves 1 y 2**.
  - Finalizando con la penúltima y última llaves (considerando que el archivo **lógico contiene una llave menos**).

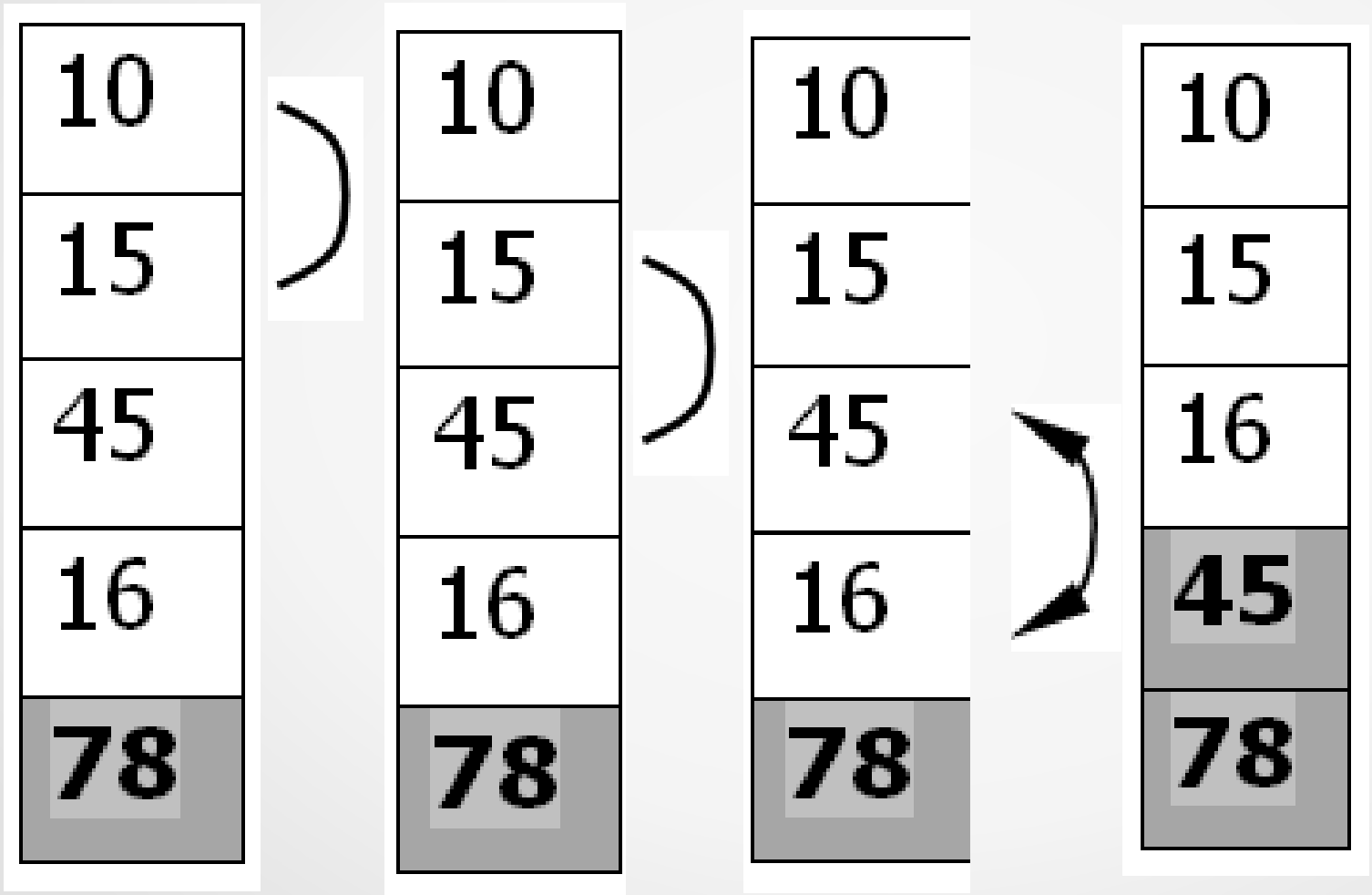
Comparación

Comparación e intercambio

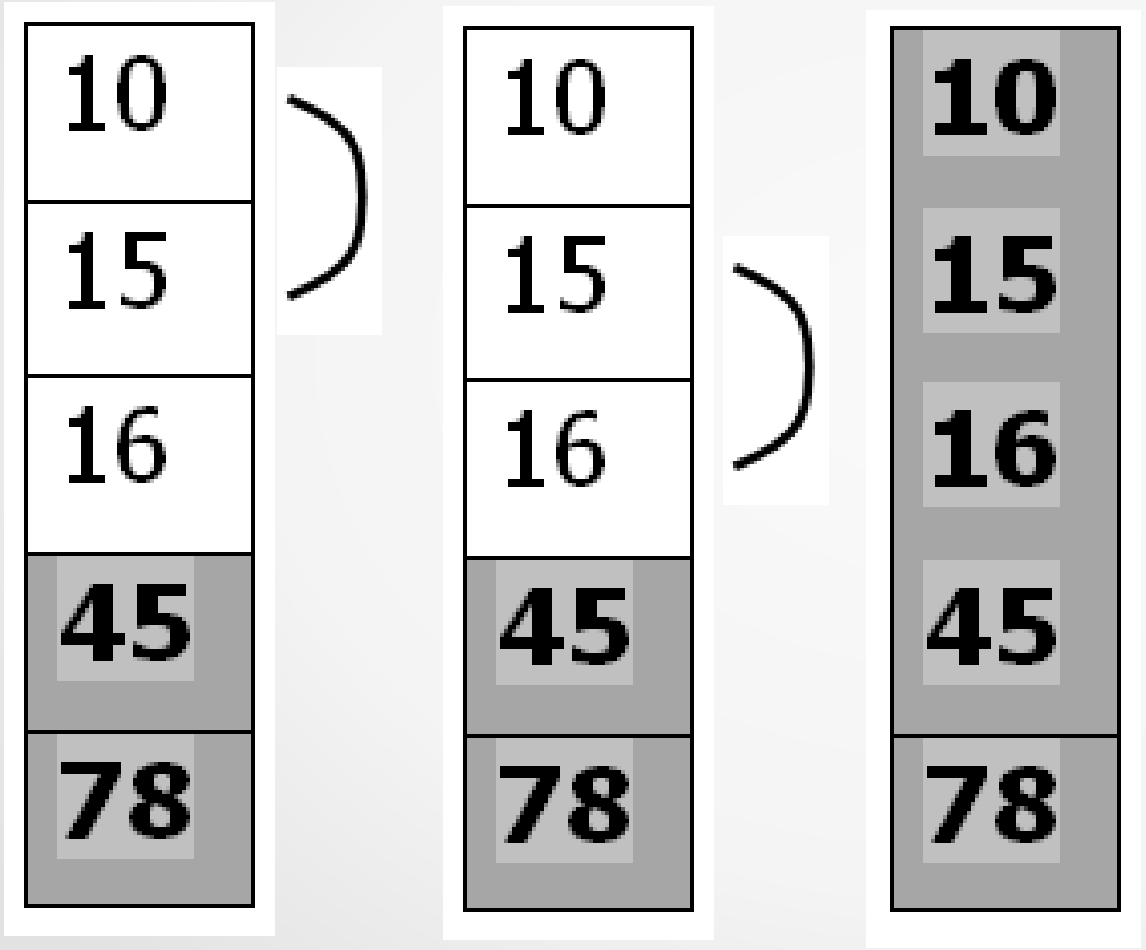
REPASO 1



REPASO 2



REPASO 3



Ejercicio 5.1

- Escriba el programa para ordenar un vector generado aleatoriamente, usando el método de la burbuja.
- Añada un contador para el número de comparaciones que hace el programa y regístrelos en una tabla como la de abajo.
- Los programas como burbuja son de orden de complejidad **$O(n^2)$**, lo que significa que la cantidad de trabajo es proporcional al cuadrado del tamaño del archivo.
- Para valores de **N** muy grandes, los programas de orden **$O(n^2)$** son intratables, por lo que hay que escribir código mas eficiente.

Tamaño del archivo (n)	Número de comparaciones
100	4,872
500	124,740
1000	499,175
5000	12,483,304
10000	49,988,784
20000	199,971,855

Ejercicio 5.1 continuación

- Hay dos alternativas para la creación del programa:
  - Escriba el programa usted mismo.
  - Solicite a cualquier herramienta de IA generativa que genere código C++ para el método de la burbuja.
    - En este último caso, si el programa generado no equivale exactamente al método que vimos en clase, usted deberá solicitar que el ciclo exterior termine si en el ciclo interior ya no se realizó ningún intercambio.
- Finalmente solicite que se añada un contador de comparaciones para medir la cantidad de trabajo realizado.

5.1.2 Quick Sort

Considerando los resultados mostrados en la tabla de la diapositiva anterior (comportamiento real del método de la burbuja), si creáramos un procedimiento que:

1. Divida los archivos grandes en cierta cantidad de *subarchivos* más pequeños.
2. Ordene los subarchivos en forma independiente.
3. Y finalmente los reúna de nuevo para obtener el archivo completo ya ordenado.

Si hacemos lo anterior, se espera que se ordene el archivo invirtiendo una menor cantidad de trabajo.

5.1.2 Quick Sort

Con base en la tabla anteriormente mostrada:

n = **10,000** comparaciones = **49,988,784**

100 subarchivos de 100 registros c/u = 10,000 registros

n = **100** comparaciones = **4,872**

100 subarchivos x 4,872 comparaciones = **487,200**

Se realiza el mismo trabajo con el 1% del esfuerzo.

5.1.2 Quick Sort

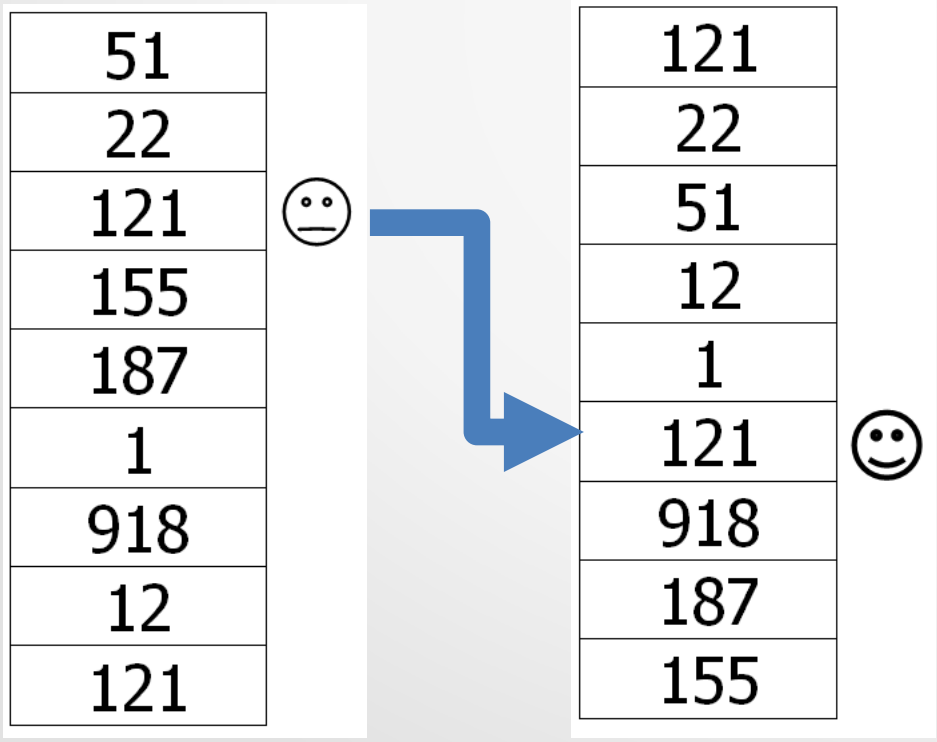
Quicksort aplica esta técnica, conocida comúnmente como "Divide y Vencerás". ¿Cómo lo hace?
Observemos un archivo de llaves diferentes **previamente ordenado.**

15
21
36
75
99
100
198
210
211

Eligiendo una llave cualquiera, se observa que todas las llaves menores a la elegida están **arriba** y las mayores **abajo**.

5.1.2 Quick Sort

Por lo tanto, si de un archivo desordenado, tomamos una **llave cualquiera** y de alguna forma, colocamos esa llave **en una posición tal** que todas las llaves menores o iguales a ella queden arriba y las mayores abajo, **sin importar el orden**, esa llave ya estará en la posición correcta.



El problema original, se transforma en dos más pequeños, un subarchivo **arriba** de la llave que quedó en su lugar y otro **abajo**.

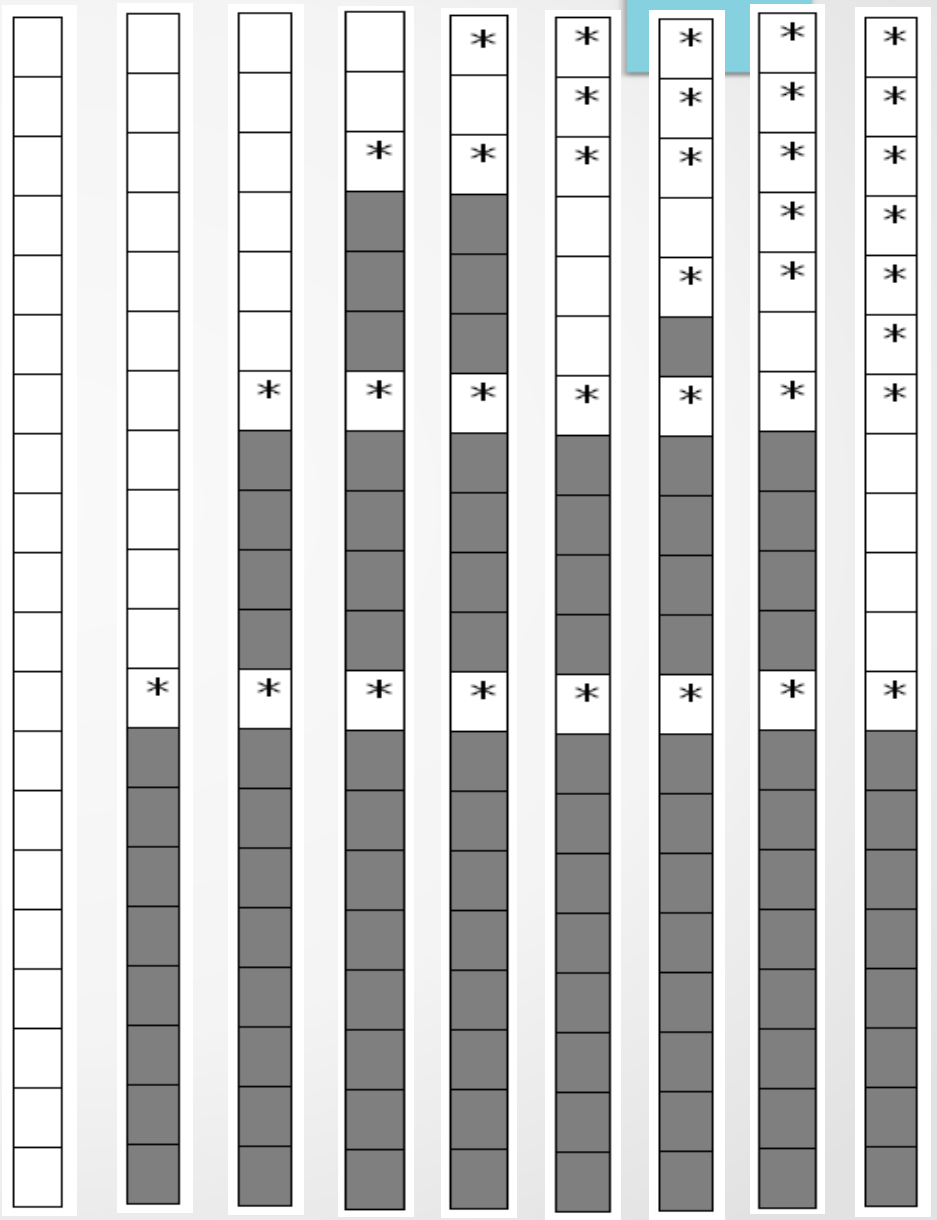
Luego se procederá a ordenar el de arriba y el otro se deja pendiente.

5.1.2 Quick Sort

- Conforme se repite este proceso, va creciendo el número de *subarchivos* por ordenar, pero los *subarchivos* van siendo cada vez más pequeños hasta llegar a tener solo un registro y, por lo tanto, ya estar ordenados.
- En cierto momento se llega a una situación en la que ya no hay más subarchivos por ordenar y el archivo completo estará ordenado.
- En la diapositiva siguiente se simula el avance del proceso.

5.1.2 Quick Sort

- Cada segmento en blanco es la parte del archivo para la cual se va a colocar en su lugar una llave.
- Los segmentos en gris indican los subarchivos pendientes.
- Los asteriscos muestran registros que ya se colocaron en su lugar.



5.1.2 Quick Sort

- **Quicksort** se reduce a un procedimiento básico que coloca en su lugar una llave.
- La ejecución repetitiva de ese procedimiento ocasionará el ordenamiento del archivo completo.

Algoritmo para colocar en su lugar un registro de un <u>subarchivo</u>.

archivo	=	nombre del archivo.
k, m	=	límites inferior y superior de un subarchivo cualquiera.
i, j	=	variables índice para encontrar el lugar.
x	=	llave del subarchivo elegida por el algoritmo, por simplicidad se elige la primera: **archivo(k)**

5.1.2 Quick Sort

1. La llave **archivo(k)** almacenada en **x** deja un **"hueco"**.

2. Las variables **i** y **j** iniciarán en los límites del subarchivo (**i** solo se incrementará, **j** solo se decrementará).

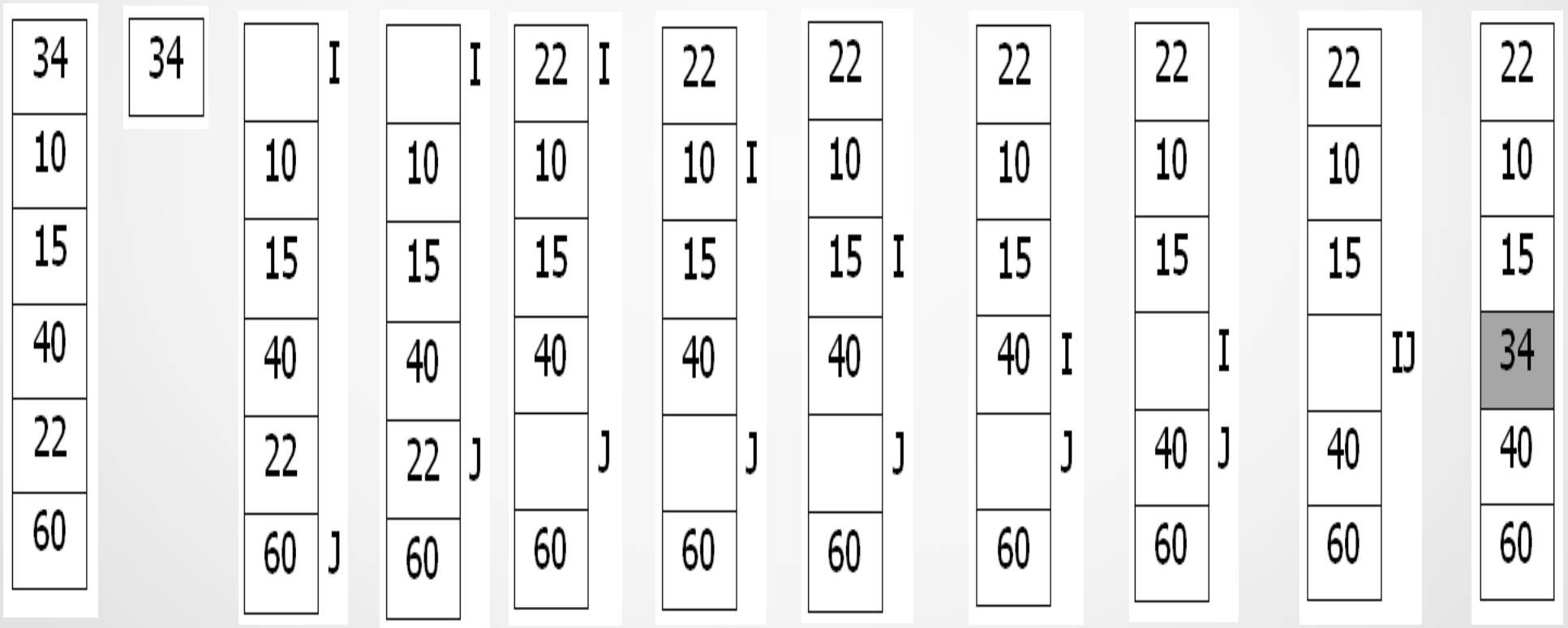
3. **Los huecos de i** deben llenarse con un valor **menor o igual que x** que encuentre **j** al decrementarse.

4. **Los huecos de j** se llenarán con un valor **mayor que x** que encuentre **i** mientras se incrementa.

5. Cuando **i y j se encuentren**, ese es el lugar que debe ocupar el valor guardado en x.

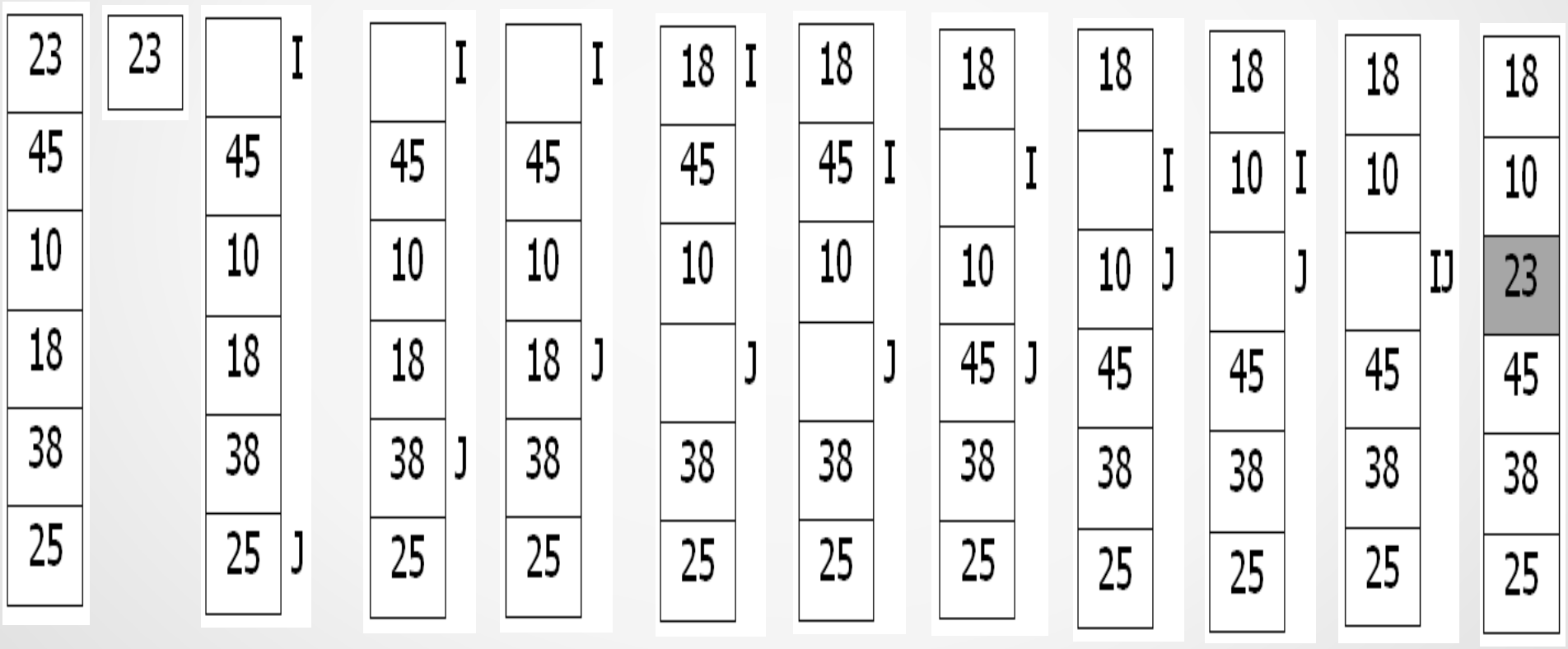
5.1.2 Quick Sort

Ejemplo de ejecución del Algoritmo



5.1.2 Quick Sort

Otro ejemplo



5.1.2 Quick Sort

Ejercicio

Escriba un programa en C++:

- Que coloque en su lugar la primera llave de un vector de tamaño N generado aleatoriamente. Inicialmente el vector completo no quedará ordenado.

- Luego se debe adecuar el programa para que usando pilas, se coloquen todos las llaves en su lugar aplicando el algoritmo básico.

```
#include <iostream>
#include "stdio.h"
#include "stdlib.h"
#include "time.h"
using namespace std;
#include "PilaDinamicaPlantilla.h"
long OrdenaUnaLlave(long[],long,long);
double NumMovimientos = 0;
int main(){
    const long n=500000;
    long archivo[n];
    srand(time(NULL));
    for(long i=0;i<n;i++) // Datos entre 1 y 10*n
        archivo[i]=rand()%(n*10)+1;
    if (n<=20){
         cout << "--Archivo original--" << endl;
         for(int i=0;i<n;i++)
            cout << archivo[i] << " ";
         cout << endl;
    }
    Pila<long> pila1,pila2;
    long k,m,posicion;
    pila1.push(0);
    pila2.push(n-1);
    do{
        k = pila1.pop();
        m = pila2.pop();
        if (k<m){
           posicion = OrdenaUnaLlave(archivo,k,m);
           pila1.push(posicion+1); pila2.push(m);
           pila1.push(k);          pila2.push(posicion-1);
        }
    }
    while (not pila1.vacia());
    if (n<=20){
           cout << "Archivo ya ordenado:" << endl;
           for(int i=0;i<n;i++)
              cout << archivo[i] << " ";
           cout << endl;
    }
    cout << "Numero de Datos         : " << n << endl;
    cout << "Numero de Movimientos   : " << NumMovimientos << endl;
    return 0;
}
```

```
long OrdenaUnaLlave(long archivo[],long k,long m){
   long temp;
   temp = archivo[k+(m-k)/2];
   archivo[k+(m-k)/2] = archivo[k];
   archivo[k]=temp;
   temp = archivo[k];
   long i = k;
   long j = m;
   bool huecoArriba = true;
   while (i<j){
      if (huecoArriba) {
         while ( ( archivo[j] > temp ) and (i<j ) )
             j--;
         if (i < j ){
             NumMovimientos++;
             archivo[ i ] = archivo[ j ];
         }
      }
      else {
         while ( ( archivo[i] <= temp ) and (i<j) )
             i++;
         if (i < j){
           NumMovimientos++;
           archivo[ j ] = archivo[ i ];
         }
      }
      huecoArriba = not huecoArriba;
    }
    archivo[ i ] = temp;
    return i;
}
```

Se intercambia el primer valor con el de la mitad para los casos en los que ya se encuentra ordenado el archivo

Quicksort iterativo

5.1.2 Quick Sort

Quicksort es un método cuyo orden de complejidad temporal es O(n log n). La tabla siguiente lo demuestra.

n	Movimientos	n * log n	Movimientos/ (n*log n)
15,000	85,285	62,641.37	1.3615
82,000	565,231	402,932.74	1.4028
100,000	698,099	500,000.00	1.3962
150,000	1,106,557	776,413.69	1.4252
300,000	2,306,304	1,643,136.38	1.4036
500,000	3,878,252	2,849,485.00	1.3610

```
#include <iostream>
#include "stdio.h"
#include "stdlib.h"
#include "time.h"
using namespace std;
void QuickSort(int[], int, int);

int main()
{
    const int n=20;
    int archivo[n];
    srand(time(NULL));
    for(long i=0;i<n;i++)
        archivo[i]=rand()%(n*10)+1;
    if (n<=20){
        cout << "--Archivo original--\n";
        for(int i=0;i<n;i++)
            cout << archivo[i] << " ";
        cout << endl;
        }

    QuickSort(archivo, 0, n-1);

    if (n<=20){
        cout << "-- Archivo ya ordenado --\n";
        for(int i=0;i<n;i++)
            cout << archivo[i] << " ";
        cout << endl;
        }

    return 0;
}
```

```
void QuickSort(int vector[], int izq, int der){
   int i, j;
   int x, t;
   i = izq; j = der;
   x = vector[(izq + der) / 2];
   while (true) {
          while (vector[i] < x)
              i++;
          while (x < vector[j])
              j--;
          if (i <= j) {
              t = vector[i];
              vector[i] = vector[j];
              vector[j]=t;
              i++;
              j--;
          }
          if (i > j)
              break;
    }
    if (izq < j) QuickSort (vector, izq, j);
    if (i < der) QuickSort (vector, i, der);
}
```

Se toma el elemento de la mitad

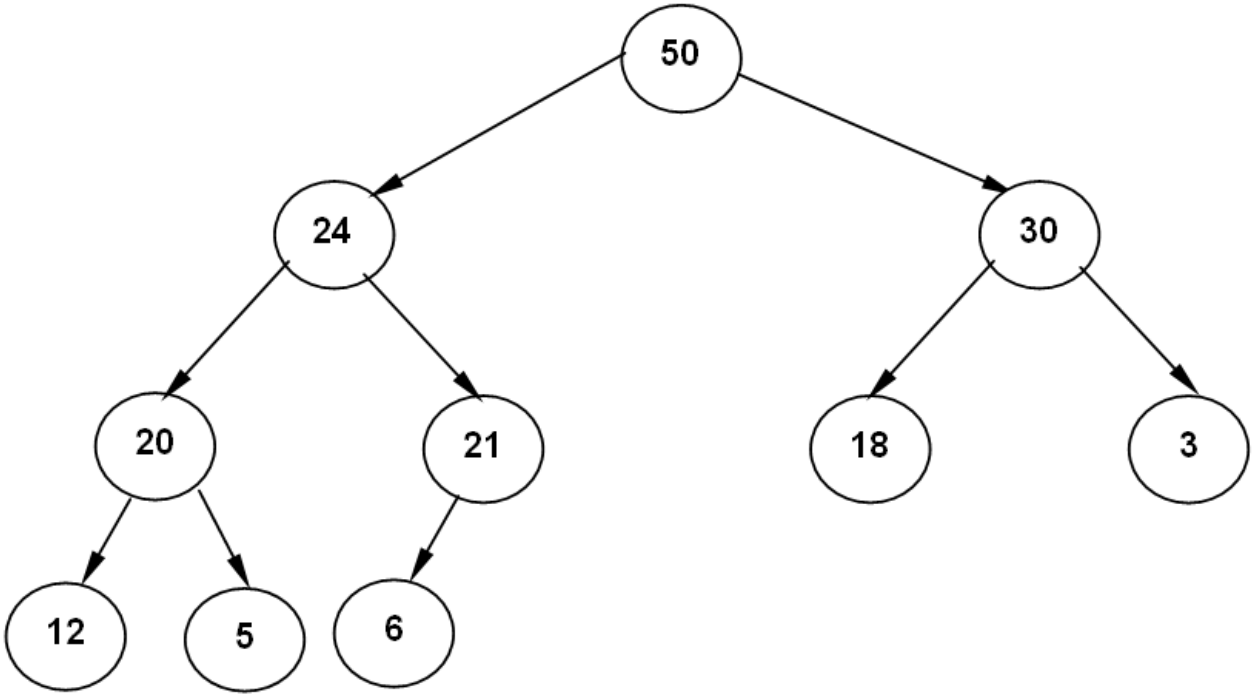
Quicksort recursivo

5.1.3 Heap Sort (Ordenamiento por Montón)

Supongamos que se cuenta con un archivo como este:

0	1	2	3	4	5	6	7	8	9
50	24	30	20	21	18	3	12	5	6

Se puede interpretar como si fuera un árbol binario:

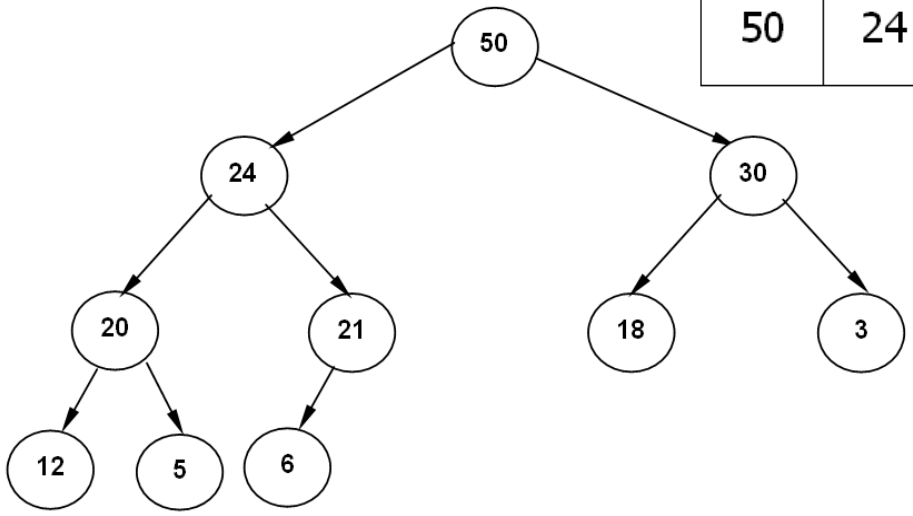


- La dirección del hijo izquierdo de un nodo de la dirección **i** es **2i+1**.
- La dirección del hijo derecho de un nodo de la dirección **i** es **2i+2**.

Si un árbol binario es un **Montón** cumplirá con las siguientes condiciones:

- Si el número total de nodos en el árbol es **impar**, todos los **nodos internos**[1] tienen dos hijos.
- Si el número total de nodos en el árbol es **par**, el último nodo interno solo tendrá un hijo (el nodo interno más extremo hacia la derecha, del penúltimo nivel).
- Las **hojas**[2] solo pueden estar en los **penúltimo y último niveles**, en caso de que las haya en el penúltimo, todas las hojas estarán **a la derecha de los nodos internos**.
- La llave de cada nodo es **mayor o igual** que las llaves de sus hijos.

[1]**nodo interno** = nodo con hijos.
[2]**hoja** = nodos sin hijos.



0	1	2	3	4	5	6	7	8	9
50	24	30	20	21	18	3	12	5	6

De acuerdo con las condiciones de arriba, el vector del ejemplo es un montón y conviene hacer una abstracción como si fuera un árbol binario como el de la izquierda.

5.1.3 Heap Sort (Ordenamiento por Montón)

Eliminación de nodos del montón

- Como lo acabamos de ver en la diapositiva anterior, el archivo del ejemplo es un Montón.
- **La eliminación** de nodos de un montón **ocasiona el ordenamiento** del archivo.
- La construcción de un Montón a partir de un archivo cualquiera la estudiaremos después.

5.1.3 Heap Sort (Ordenamiento por Montón)

Eliminar un nodo de un Montón consiste en:

- Retirar la llave de la raíz (la mayor del montón).
- Reubicar el resto de las llaves de manera que la estructura siga siendo un montón.
- Para hacerlo, se debe eliminar un nodo.
- El nodo elegido es el único que se permite para que la estructura siga siendo montón: la última hoja (extrema derecha del último nivel).
- El único lugar disponible para colocar la llave del nodo eliminado es el lugar vacante dejado por la raíz.
- Una vez ahí, la llave debe ir bajando hasta el lugar en que su valor sea mayor que el de sus hijos.

5.1.3 Heap Sort (Ordenamiento por Montón)

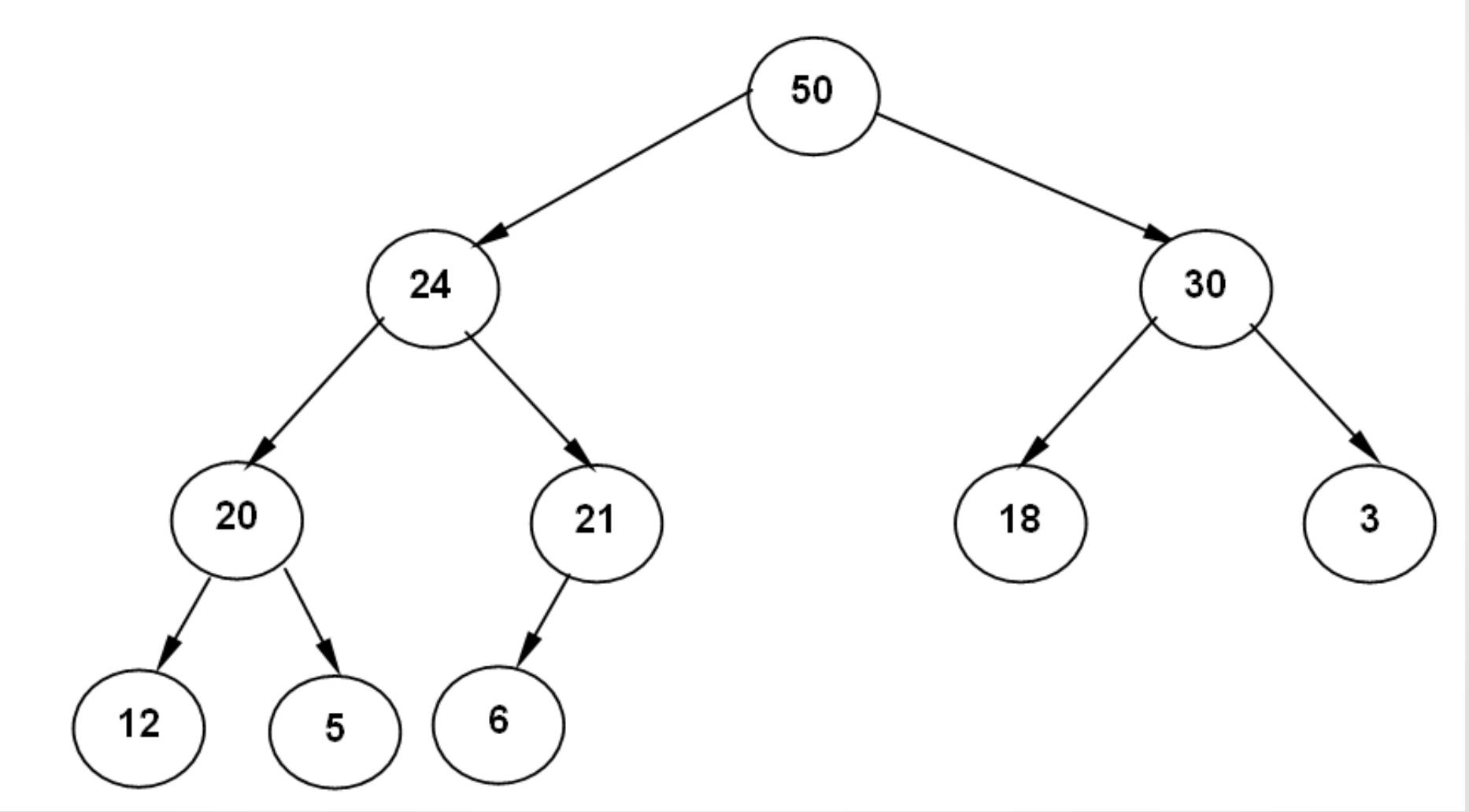
El algoritmo para bajar la llave que estaba en la última hoja (llamémosla K) es el siguiente:

1. Si K es mayor o igual que el mayor de sus hijos, está en el lugar adecuado y el proceso termina.
2. De lo contrario, intercambia su lugar con el mayor de sus hijos.
3. Se repite el paso 1.

Este algoritmo requiere que existan al menos dos nodos en el árbol.

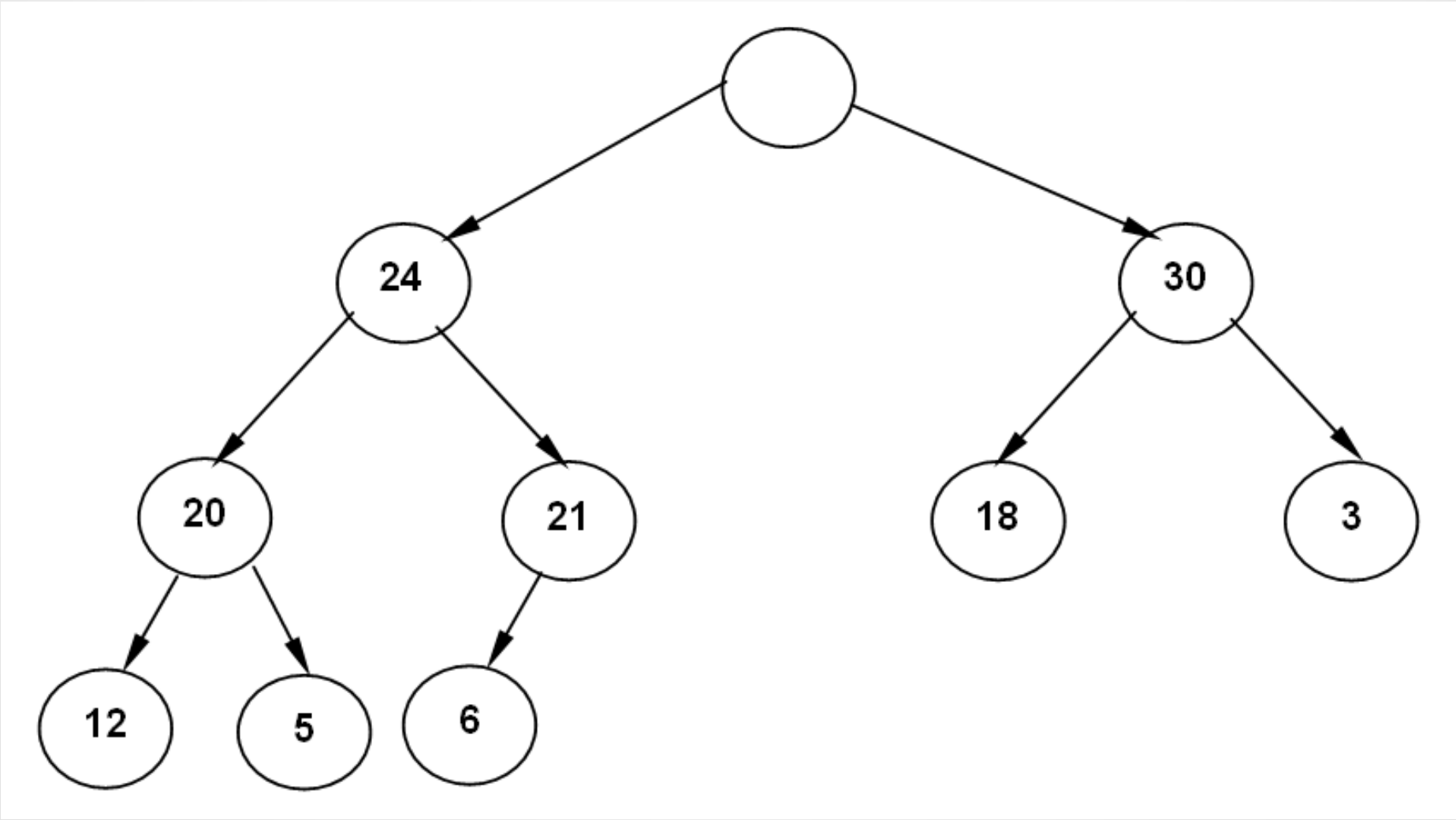
5.1.3 Heap Sort (Ordenamiento por Montón)

Se removerá un nodo del ejemplo



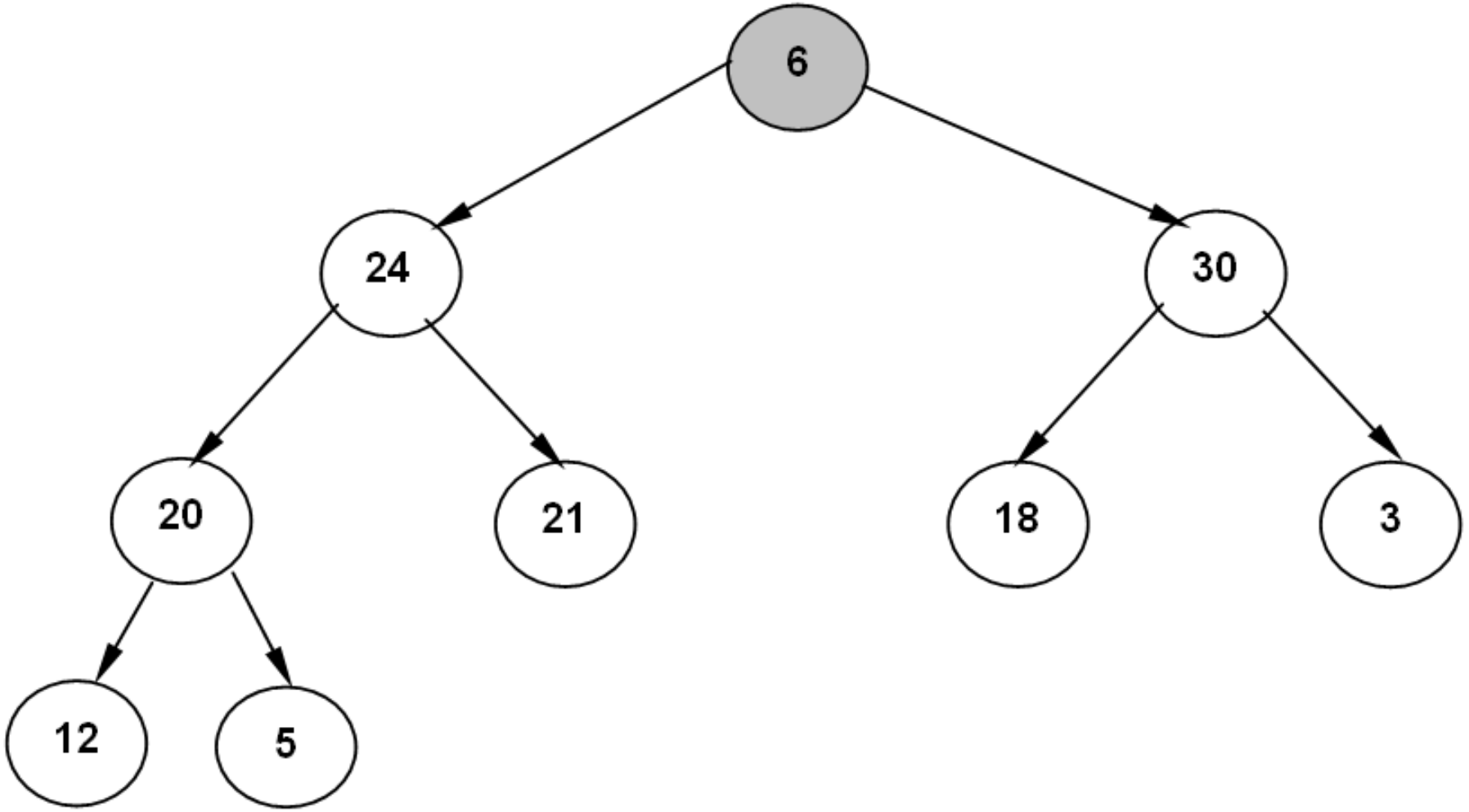
5.1.3 Heap Sort (Ordenamiento por Montón)

Sale la llave de la raíz



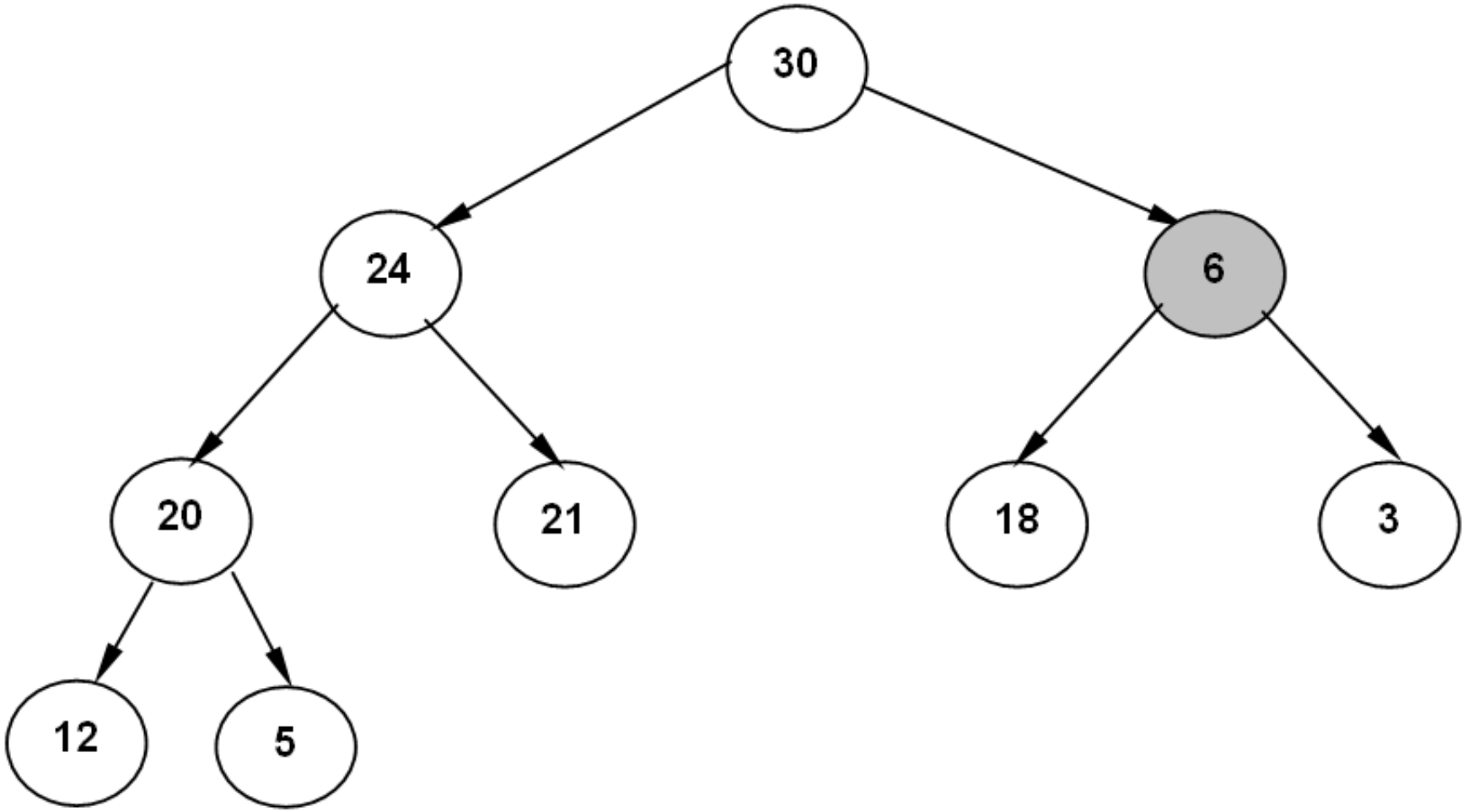
5.1.3 Heap Sort (Ordenamiento por Montón)

Se intenta colocar el valor de la última hoja en la raíz. Como no cumple con la condición de que la llave sea mayor que las de sus hijos, se deberá intercambiar con la mayor de sus hijos

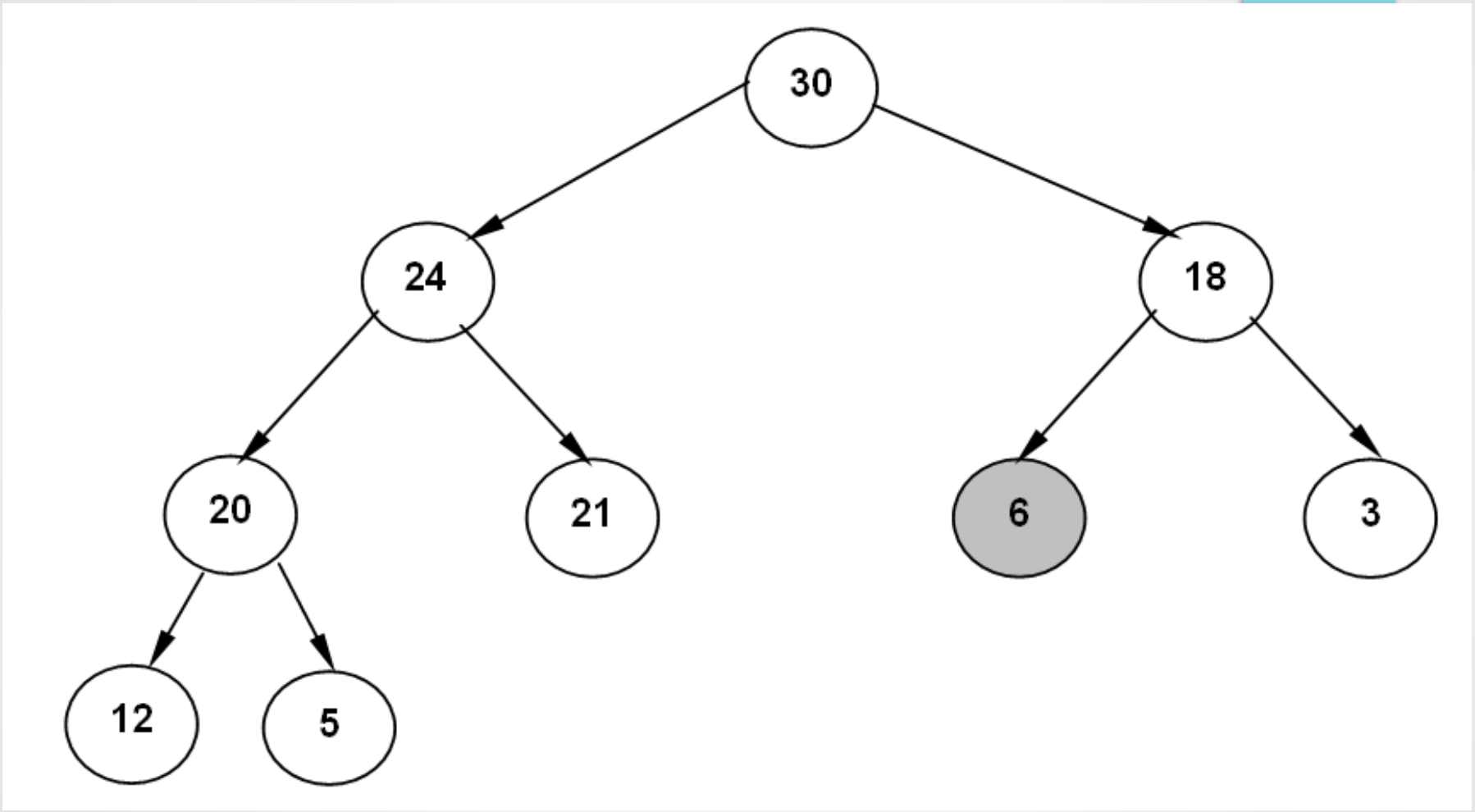


5.1.3 Heap Sort (Ordenamiento por Montón)

Sigue sin cumplir con la condición por lo que se deberá intercambiar con el mayor de sus hijos



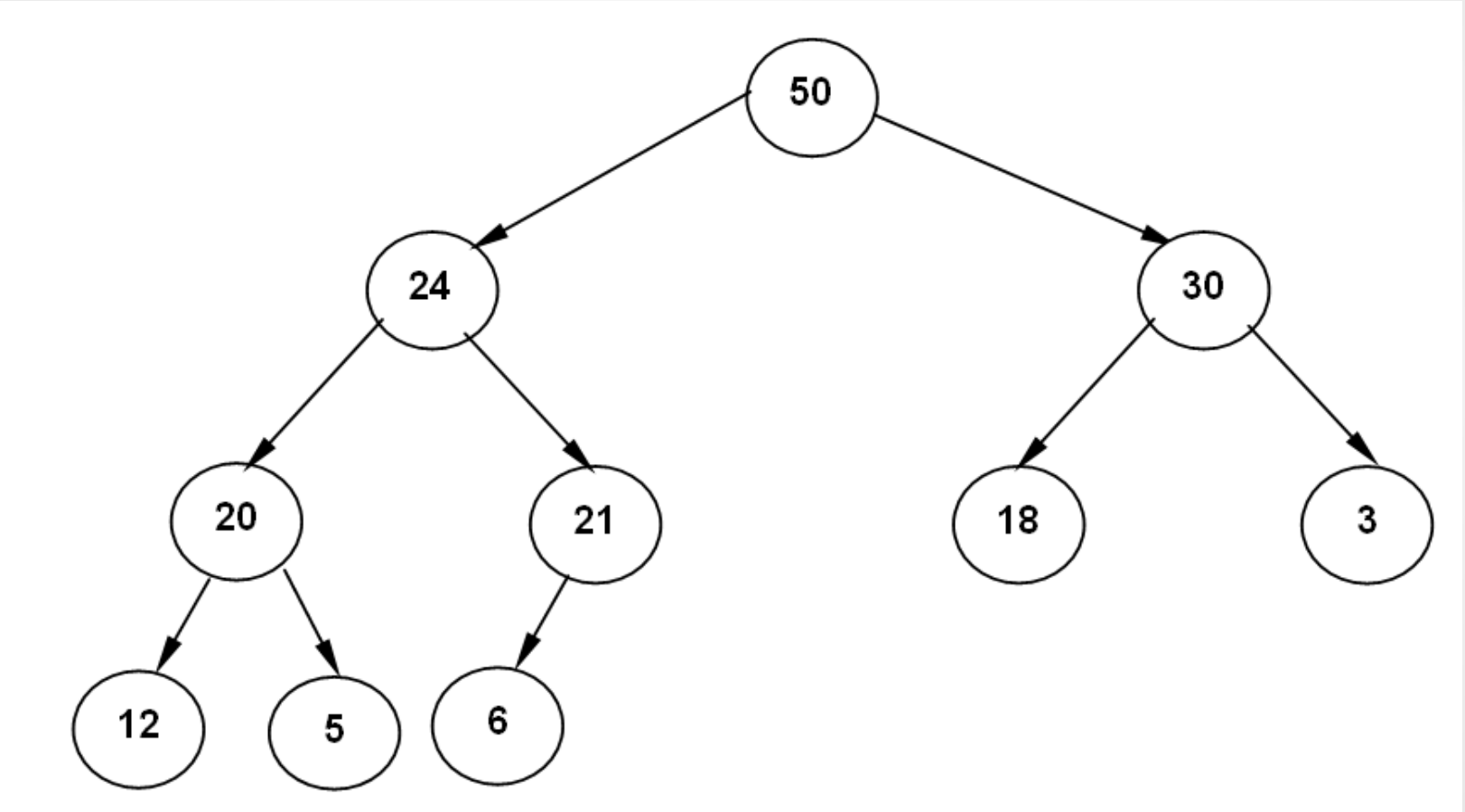
5.1.3 Heap Sort (Ordenamiento por Montón)



La llave, al llegar a un nodo que es una hoja, se considera cumple con la condición porque no tiene hijos

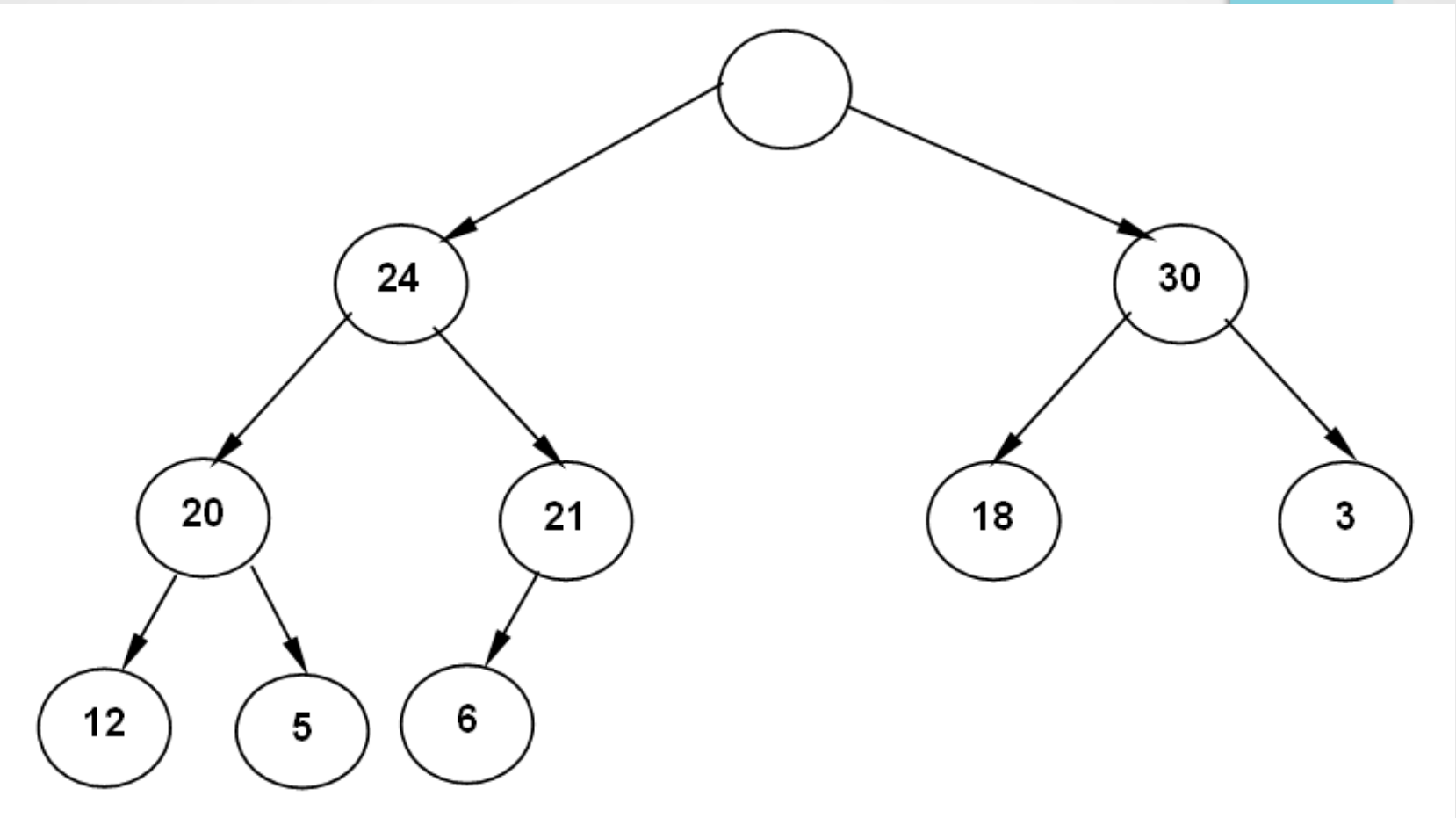
5.1.3 Heap Sort (Ordenamiento por Montón)

Repasemos el proceso



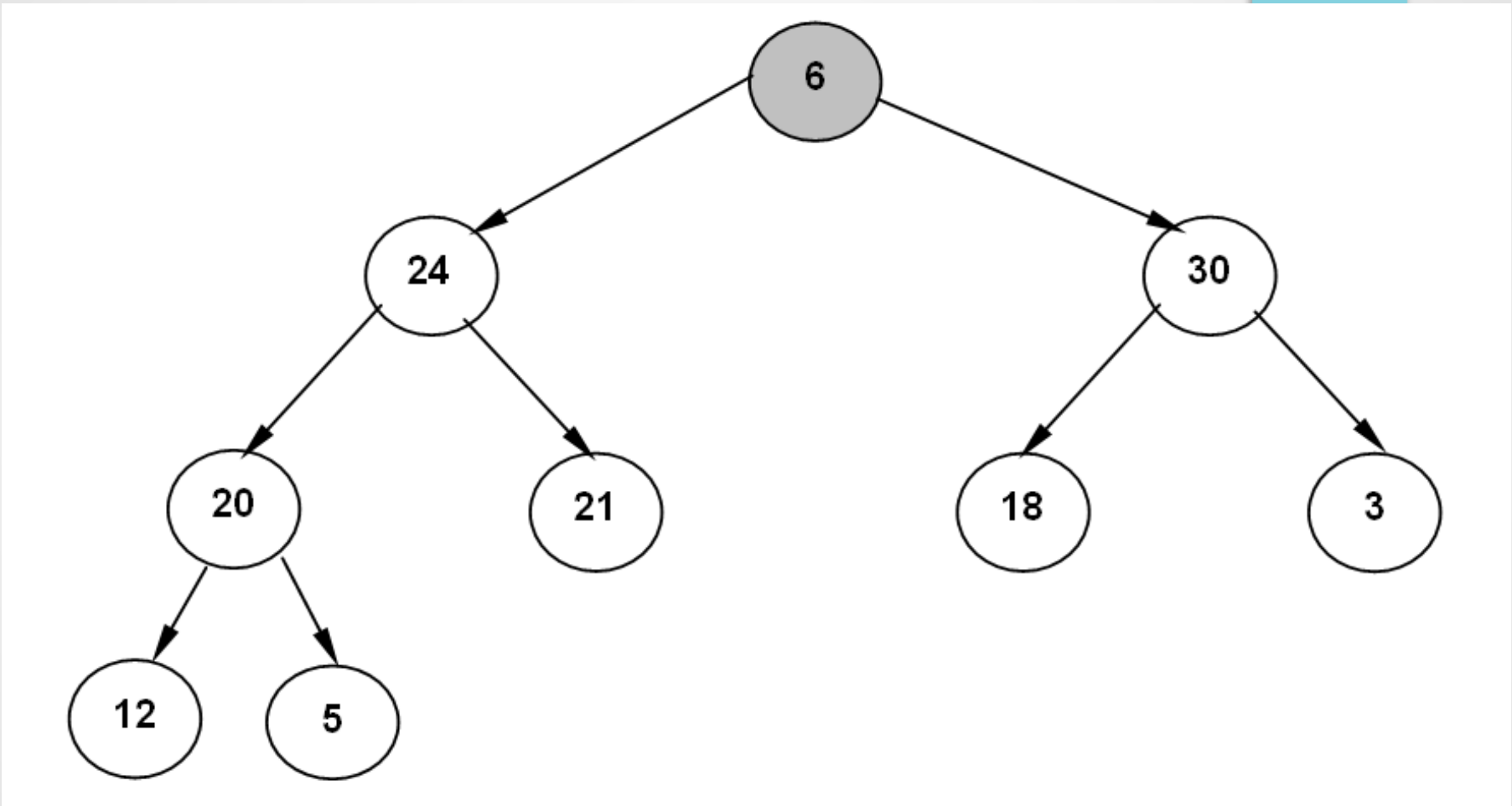
50	24	30	20	21	18	3	12	5	**6**

5.1.3 Heap Sort (Ordenamiento por Montón)



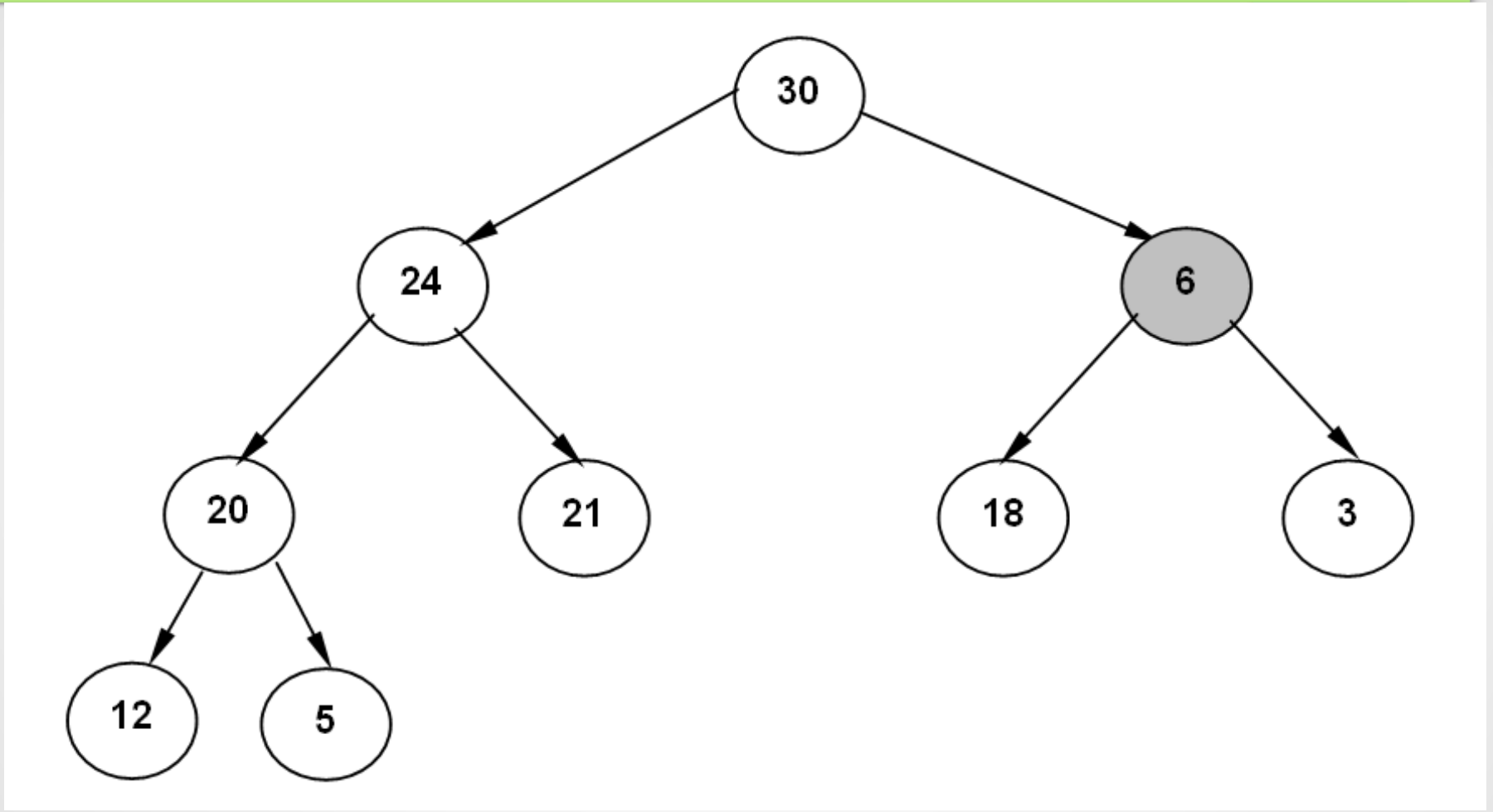
	24	30	20	21	18	3	12	5	**6**

5.1.3 Heap Sort (Ordenamiento por Montón)



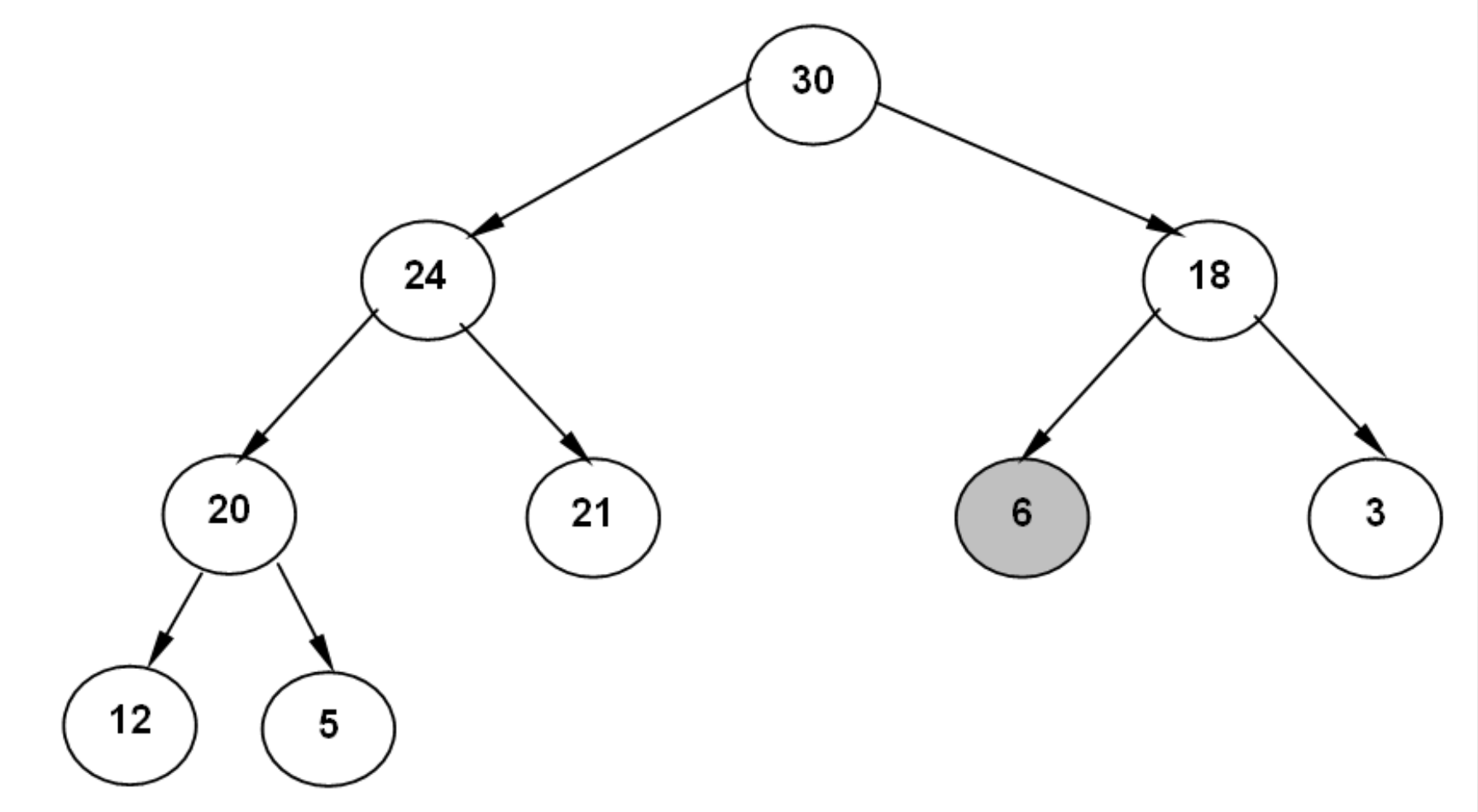
6	24	30	20	21	18	3	12	5	50

5.1.3 Heap Sort (Ordenamiento por Montón)



30	24	6	20	21	18	3	12	5	50

5.1.3 Heap Sort (Ordenamiento por Montón)



30	24	18	20	21	**6**	3	12	5	**50**

5.1.3 Heap Sort (Ordenamiento por Montón)

Ejercicio

A partir del vector siguiente (que ya es un montón), ejecute la eliminación de nodos hasta que el vector quede completamente ordenado.

94	68	80	26	35	51	21	3	8	18	10	42	15

5.1.3 Heap Sort (Ordenamiento por Montón)

Del procedimiento que se acaba de conocer, se desprenden los siguientes fundamentos para la CONSTRUCCIÓN DE UN MONTÓN

- A partir de un nodo que tiene como hijos a las raíces de dos montones, se puede construir un nuevo montón.
- Cada hoja es un montón.

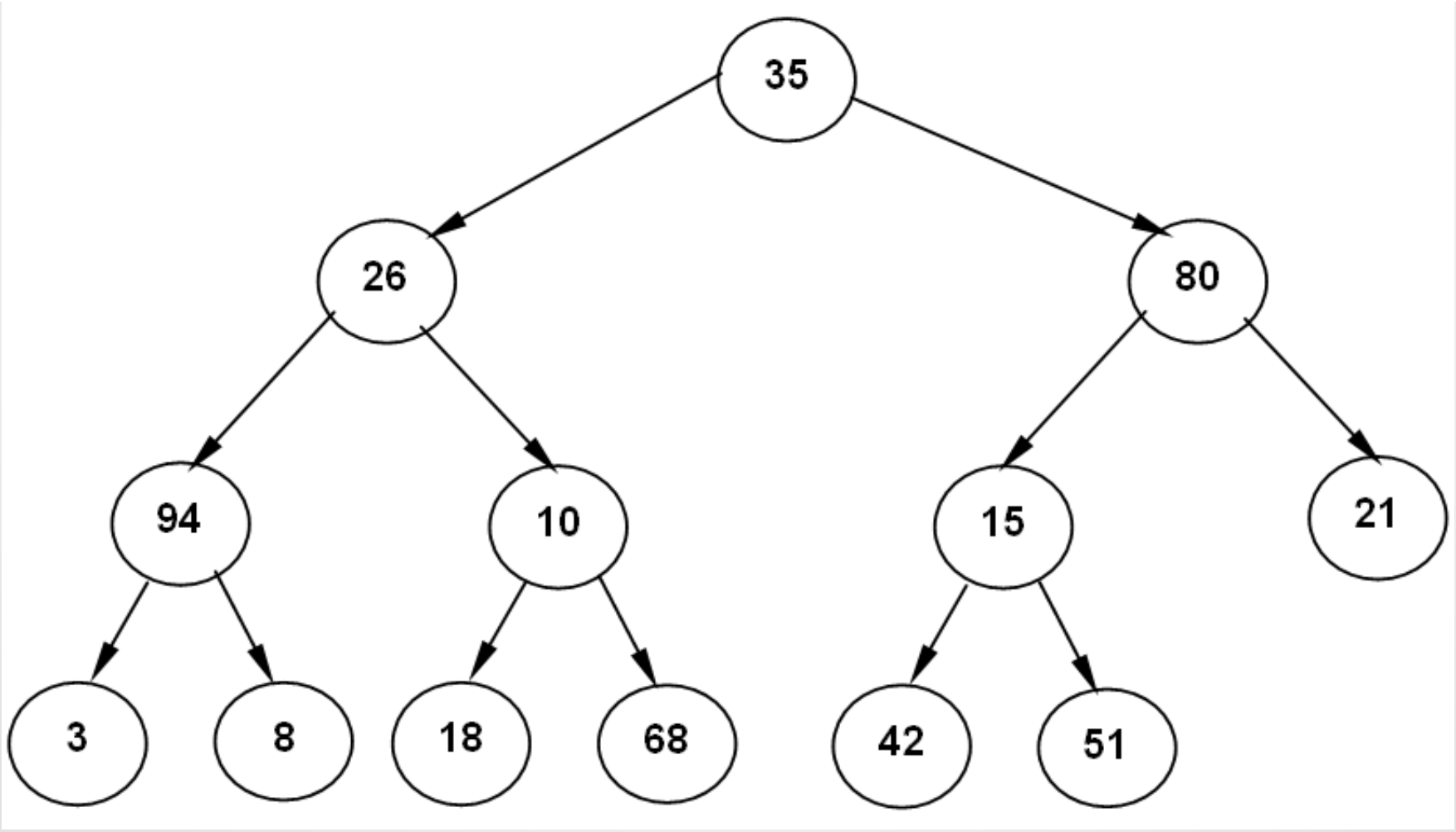
5.1.3 Heap Sort (Ordenamiento por Montón)

Algoritmo para la Construcción:

- Se inicia el proceso con el último nodo interno (penúltimo nivel).
  - Se debe verificar si su llave es mayor que la de sus hijos.
  - Si cumple con esa condición, significa que la llave está en su lugar.
  - Si no cumple con la segunda condición, se debe bajar la llave hasta el lugar adecuado.
- El proceso continúa con el siguiente nodo a la izquierda, si ya terminó un nivel, se pasa al nivel anterior con el primer nodo desde la derecha.
- Una vez terminado el proceso, al llegar a la raíz, el archivo completo es un montón.

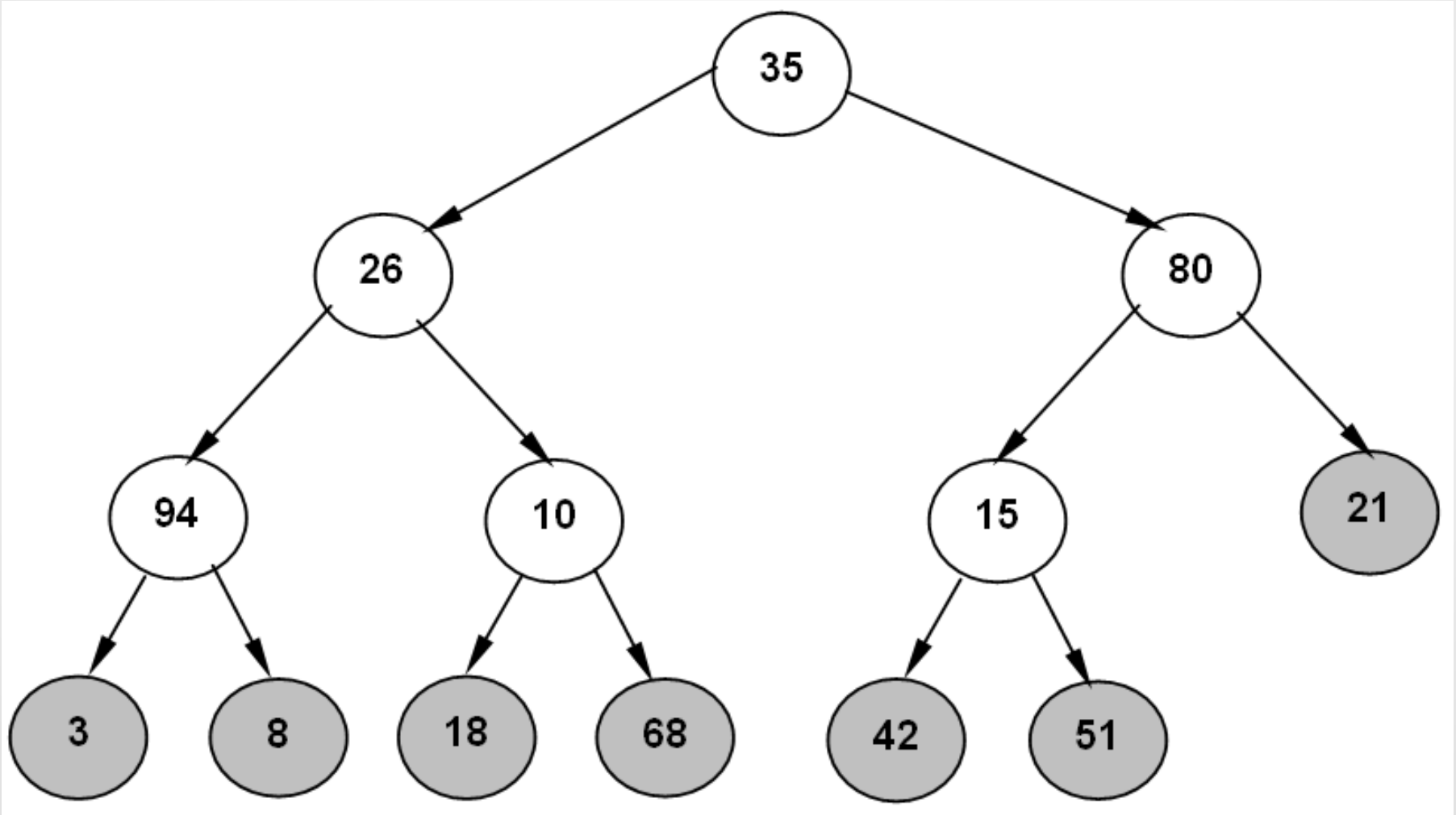
5.1.3 Heap Sort (Ordenamiento por Montón)

35	26	80	94	10	15	21	3	8	18	68	42	51

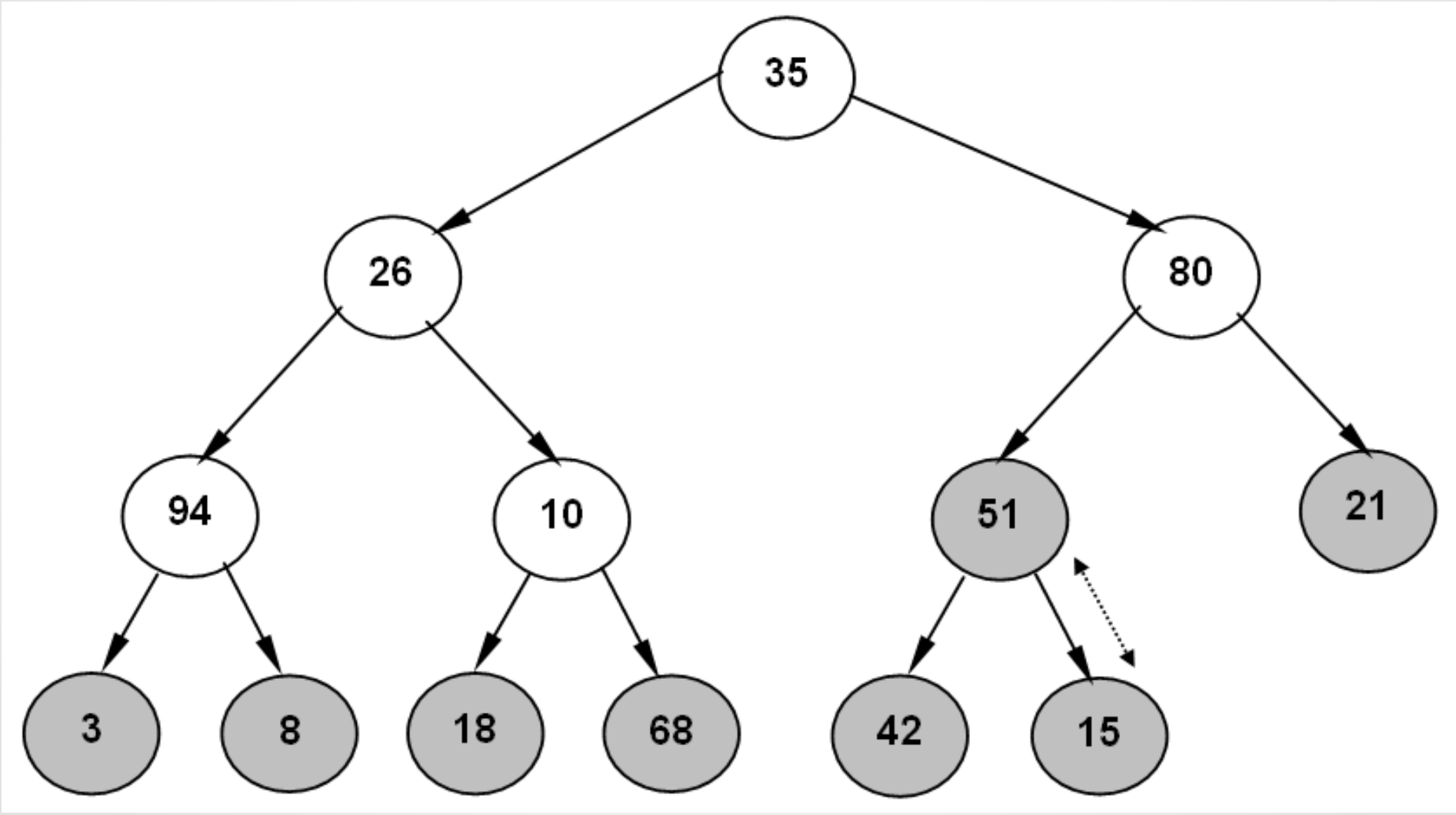


5.1.3 Heap Sort (Ordenamiento por Montón)

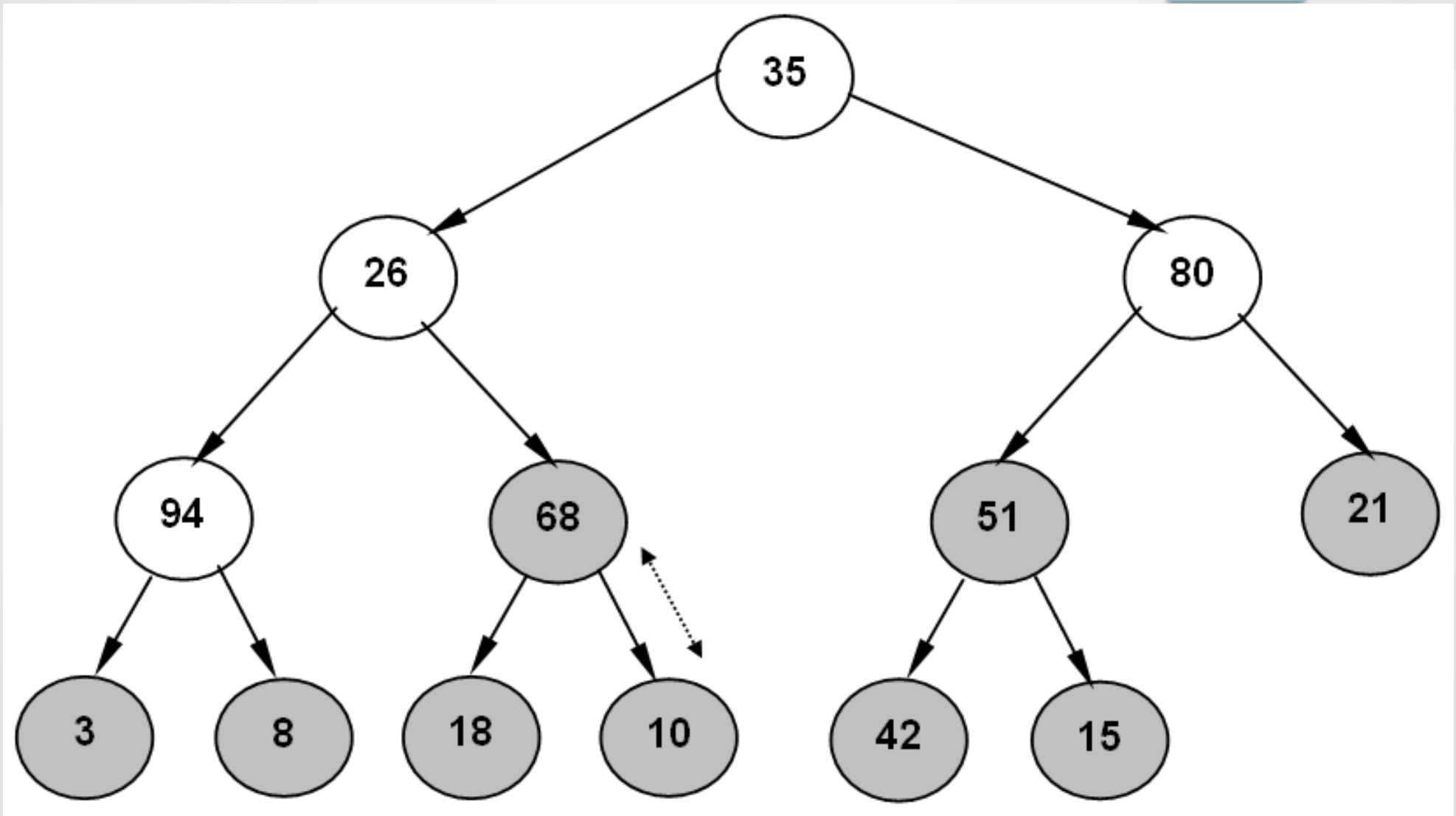
Cada nodo sombreado representa a la raíz de un montón



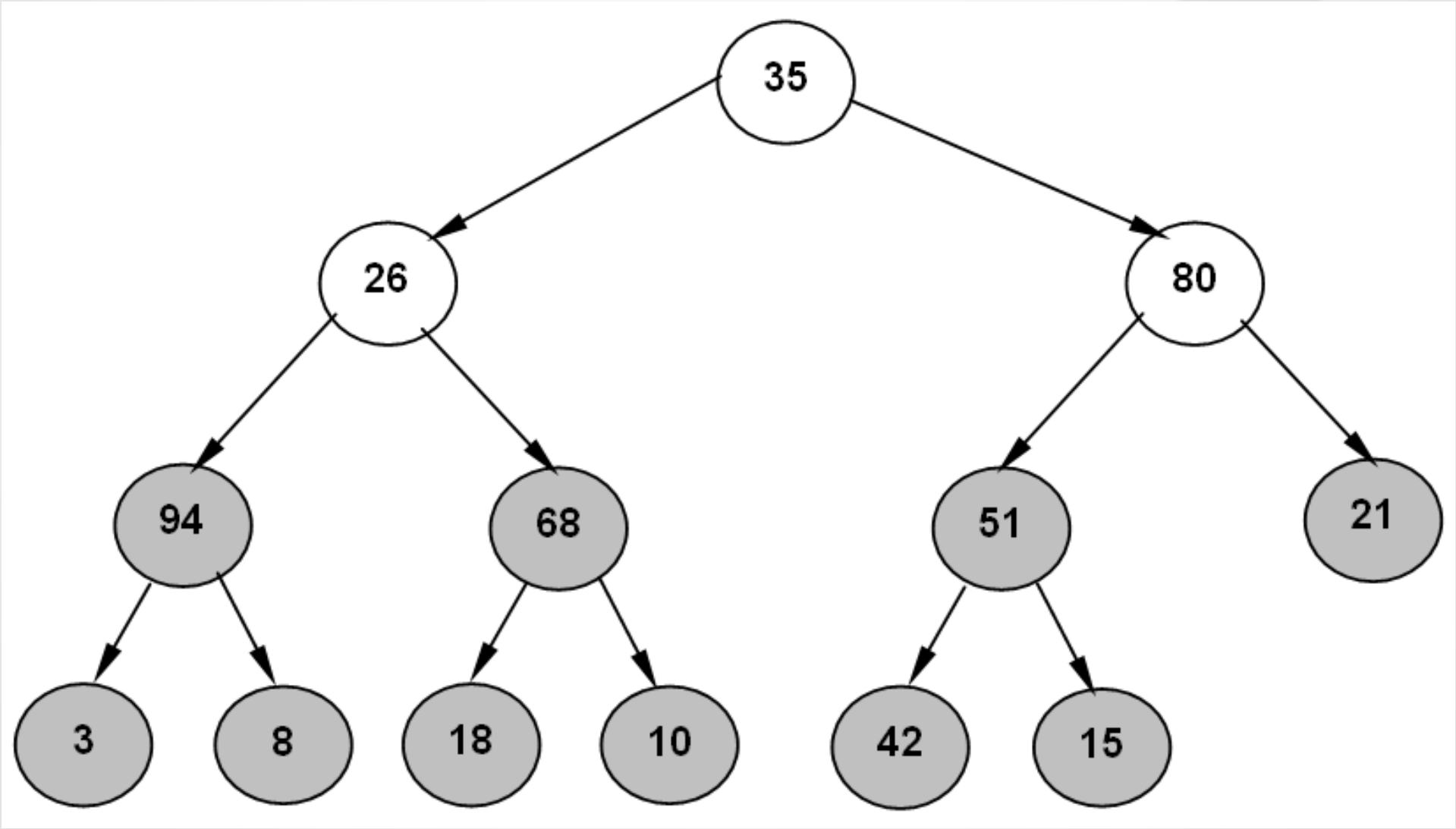
5.1.3 Heap Sort (Ordenamiento por Montón)



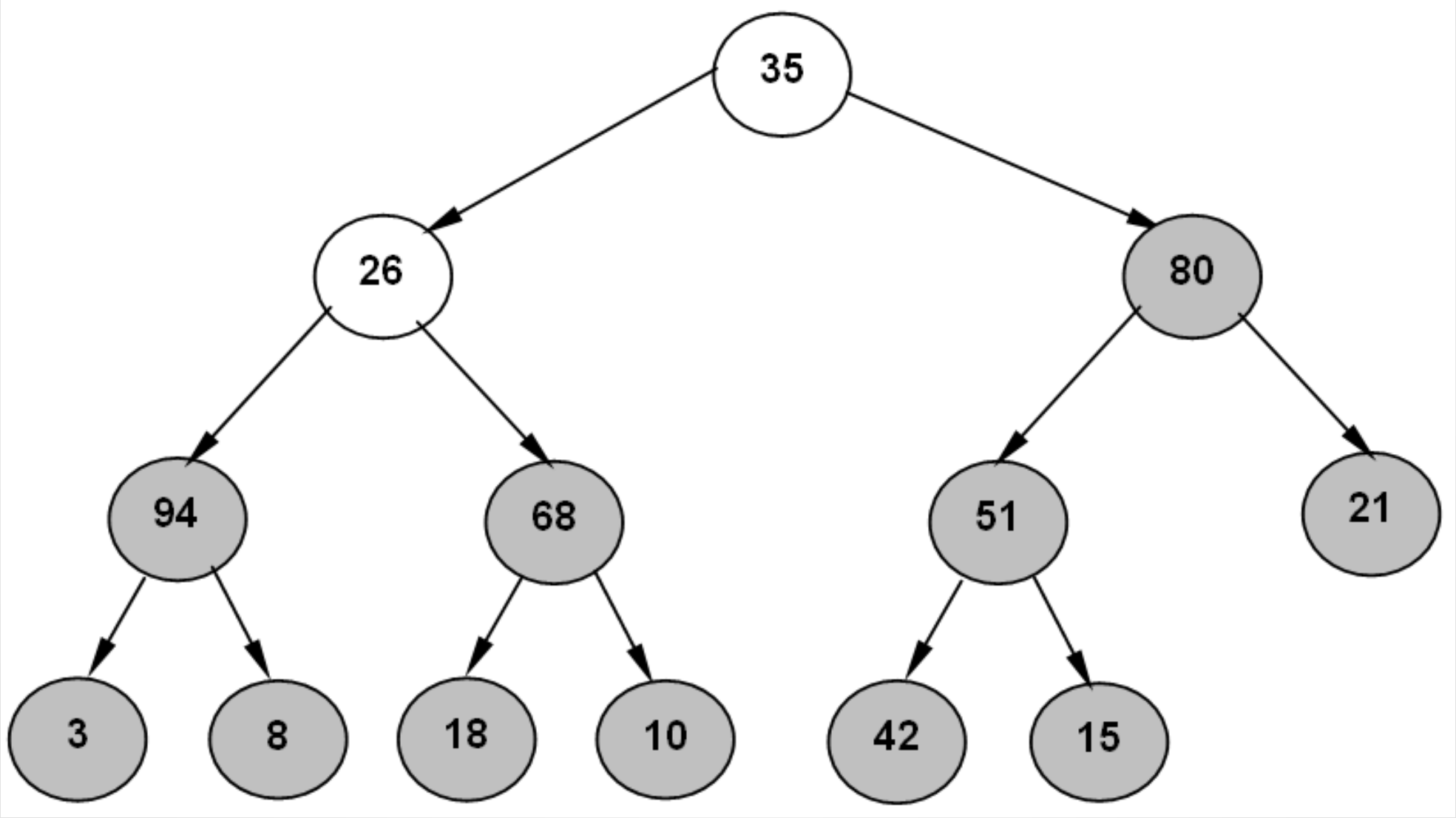
5.1.3 Heap Sort (Ordenamiento por Montón)



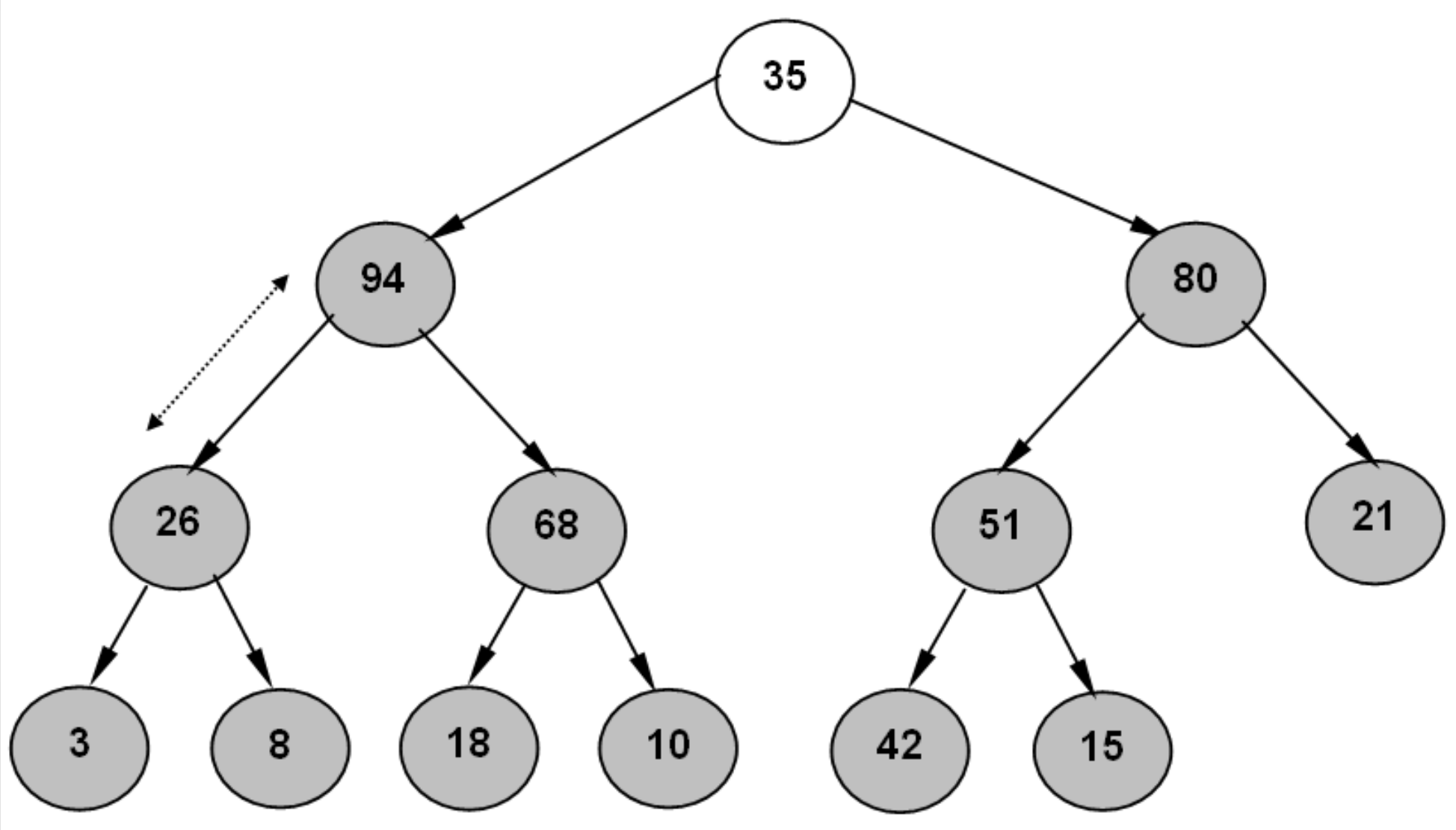
5.1.3 Heap Sort (Ordenamiento por Montón)



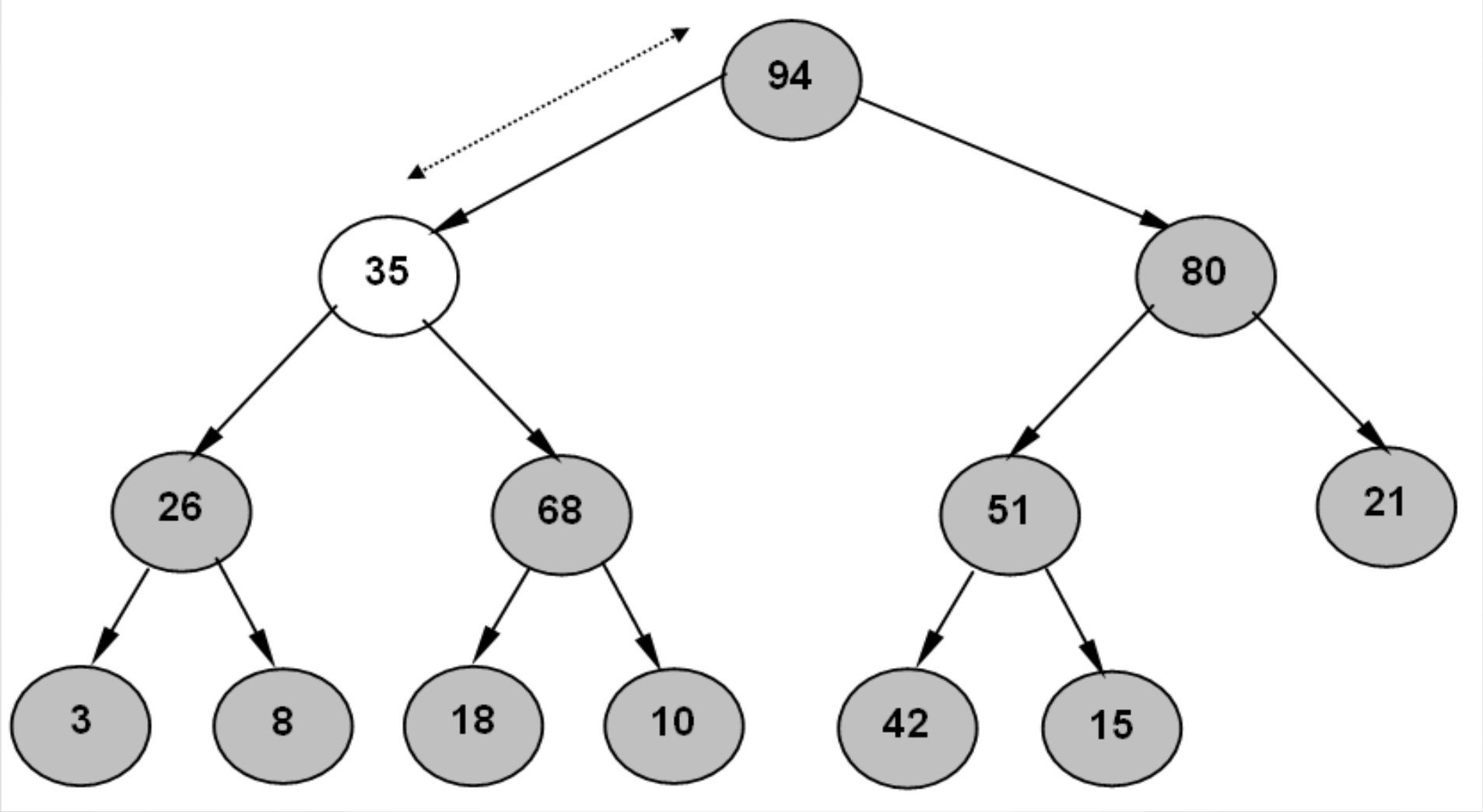
5.1.3 Heap Sort (Ordenamiento por Montón)



5.1.3 Heap Sort (Ordenamiento por Montón)

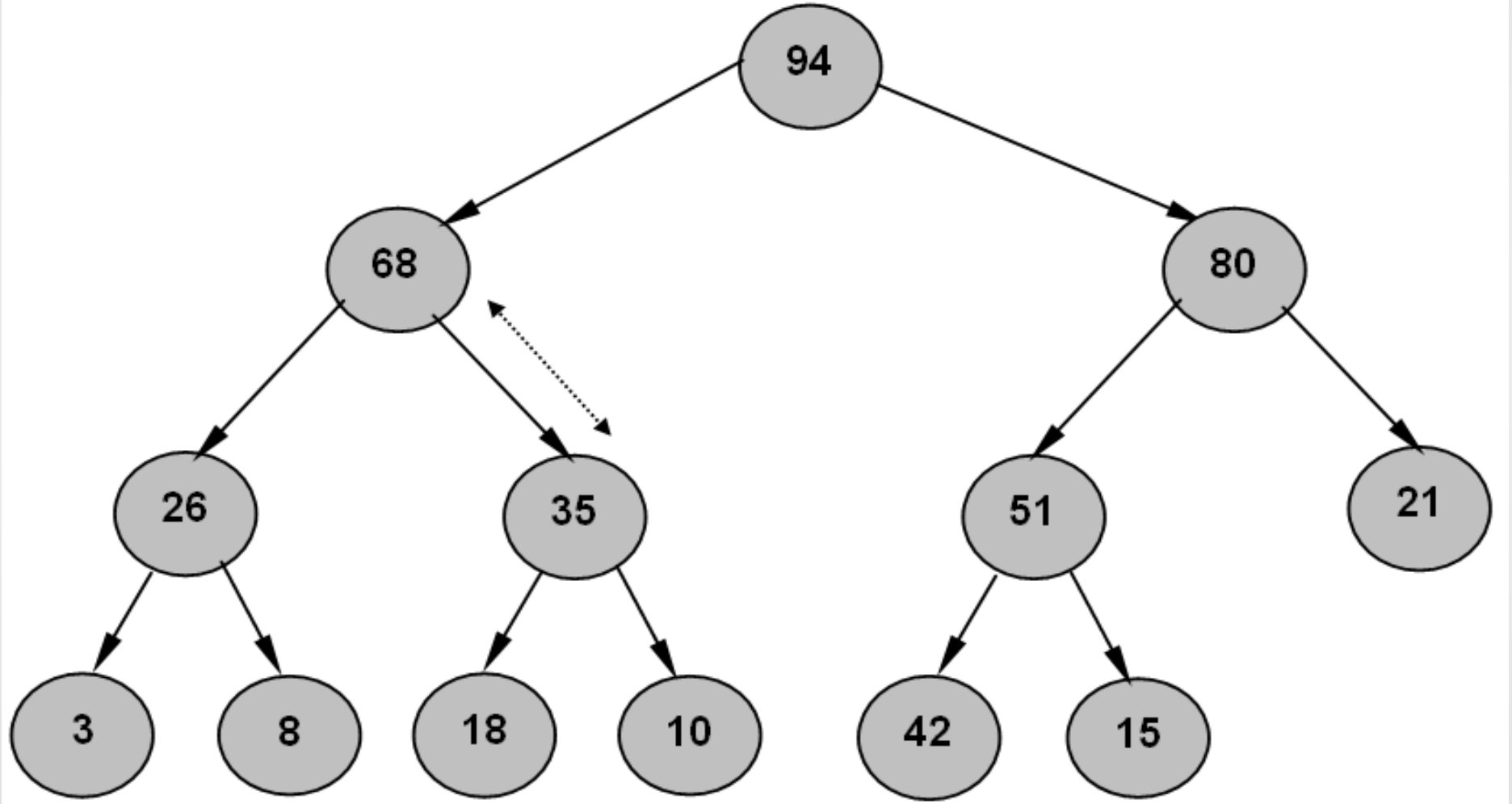


5.1.3 Heap Sort (Ordenamiento por Montón)



5.1.3 Heap Sort (Ordenamiento por Montón)

El archivo ya es un montón



Animacion de Heap Sort:

https://www.cs.usfca.edu/~galles/visualization/HeapSort.html

5.1.3 Heap Sort (Ordenamiento por Montón)

Observaciones de utilidad para la programación

Si n es par	el último nodo interno tiene un hijo
Si n es impar	el último nodo interno tiene dos hijos

Posiciones importantes en el vector

0	dirección de la raíz
n-1	dirección de la última hoja
int(n/2)-1	dirección del último nodo interno
2i+1	dirección del hijo izquierdo del nodo i
2i+2	dirección del hijo derecho del nodo i

Programa iterativo para la Construcción de un Montón

```
#include <iostream>
using namespace std;
#include "stdio.h"
#include "stdlib.h"
#include "time.h"
double NumIntercambios = 0;
int main()
{
    const long n=15;
    long vector[n];
    srand(time(NULL)); // semilla
    for(long i=0;i<n;i++) // Genera Datos al Azar
        vector[i]=rand()%100+1;
    if (n<=20){
            cout << "--Archivo original--" << endl;
            for(long i=0;i<n;i++)
                    cout << vector[i] << " ";
            cout << endl;
    }
    for(long i=int(n/2)-1;i>=0;i--){
            long RaizMonton = i;
            do{
                    long HijoIzq = RaizMonton*2+1;
                    long HijoDer = RaizMonton*2+2;
                    long HijoMayor;
                    if (RaizMonton>int(n/2)-1)
                            break;
                    if ((RaizMonton==int(n/2)-1) and (int(n/2)==n*1.0/2))
                            HijoMayor = HijoIzq;
                    else if (vector[HijoIzq]>vector[HijoDer])
                            HijoMayor = HijoIzq;
                    else
                            HijoMayor = HijoDer;
                    if (vector[RaizMonton]<vector[HijoMayor]){
                            long temp=vector[RaizMonton];
                            vector[RaizMonton]=vector[HijoMayor];
                            vector[HijoMayor]=temp;
                            RaizMonton = HijoMayor;
                            NumIntercambios++;
                    }
                    else break;
            }while (true);
    }
    if (n<=20){
            cout << "--Archivo Monton--" << endl;
            for(long i=0;i<n;i++)
                    cout << vector[i] << " ";
            cout << endl;
    }
    printf("\nNumero de Intercambios: %f\n",NumIntercambios);
    return 0;
}
```

5.1.3 Heap Sort (Ordenamiento por Montón)

Cantidad de trabajo realizado por HeapSort (solo la construcción del montón) y comprobación del orden de complejidad temporal n log n

n	Número de intercambios	n*logn	Intercambios / (n*logn)
10,000	7,386	40,000.00	0.1847
50,000	37,115	234,948.50	0.1580
100,000	73,819	500,000.00	0.1476
200,000	148,118	1,060,206.00	0.1397
350,000	258,096	1,940,423.82	0.1330
500,000	369,440	2,849,485.00	0.1297

5.1.4 Inserción Simple

Se basa en la premisa de que un archivo que contiene solo un registro ya está ordenado.

Razonamiento:

Se supone que el primer registro de un archivo de tamaño **n**, conforma un subarchivo ya ordenado y los restantes **n–1** registros forman un archivo desordenado.

5.1.4 Inserción Simple

Archivo Original

34
10
15
40
22
20

Según el razonamiento de Inserción Simple

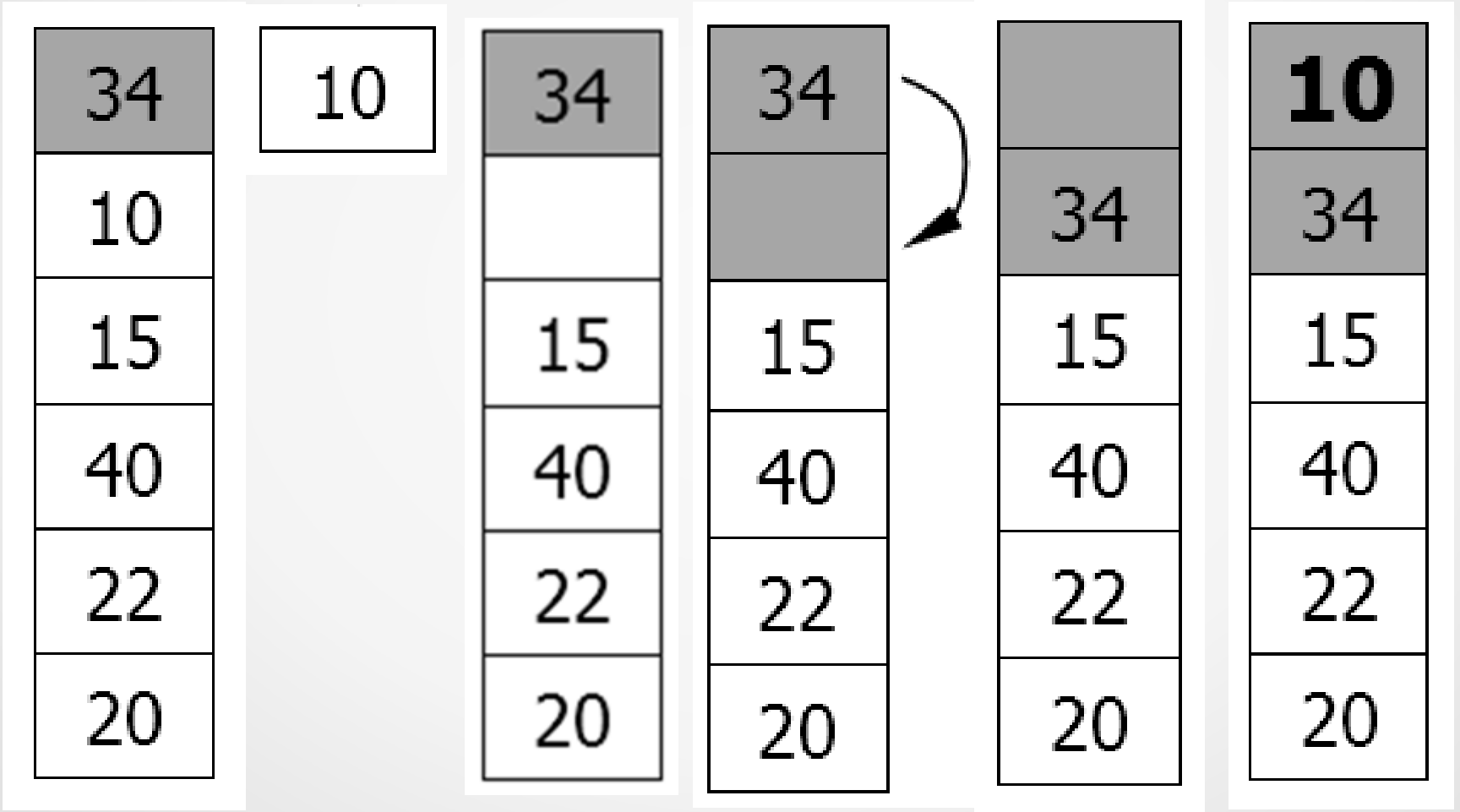
34	Subarchivo ordenado
10	Subarchivo desordenado
15	
40	
22	
20	

5.1.4 Inserción Simple

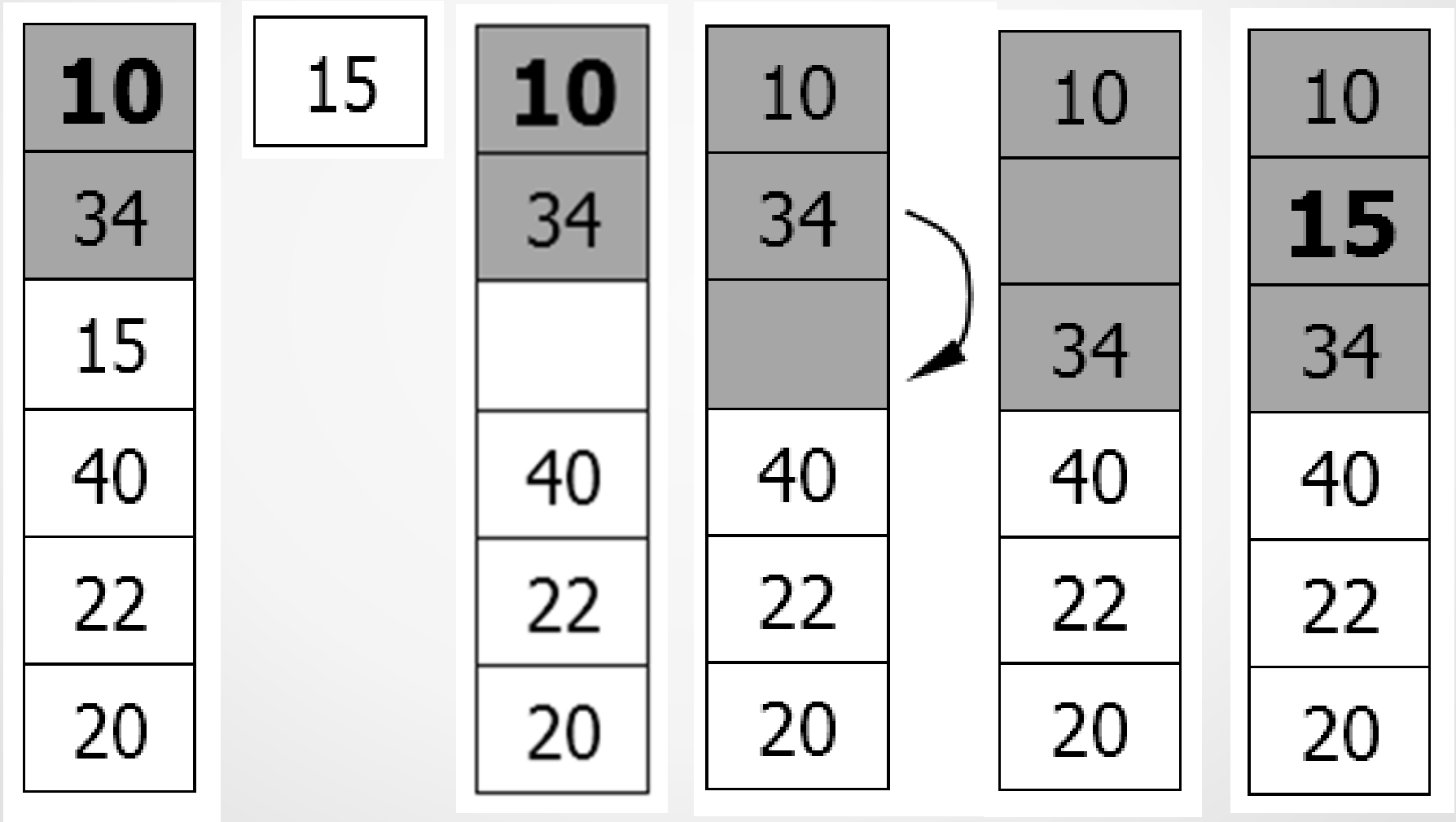
Procedimiento

- Se toma el primer registro del subarchivo desordenado y se coloca dentro del subarchivo ordenado que ahora tendrá un registro más permaneciendo ordenado.
- El subarchivo desordenado reducirá su tamaño en **1** registro.
- Los procesos anteriores se repiten hasta que el subarchivo ordenado sea de tamaño **n** y el desordenado desaparezca.
- El nombre del método se toma del proceso de **insertar** uno de los registros del segmento desordenado dentro del ordenado.

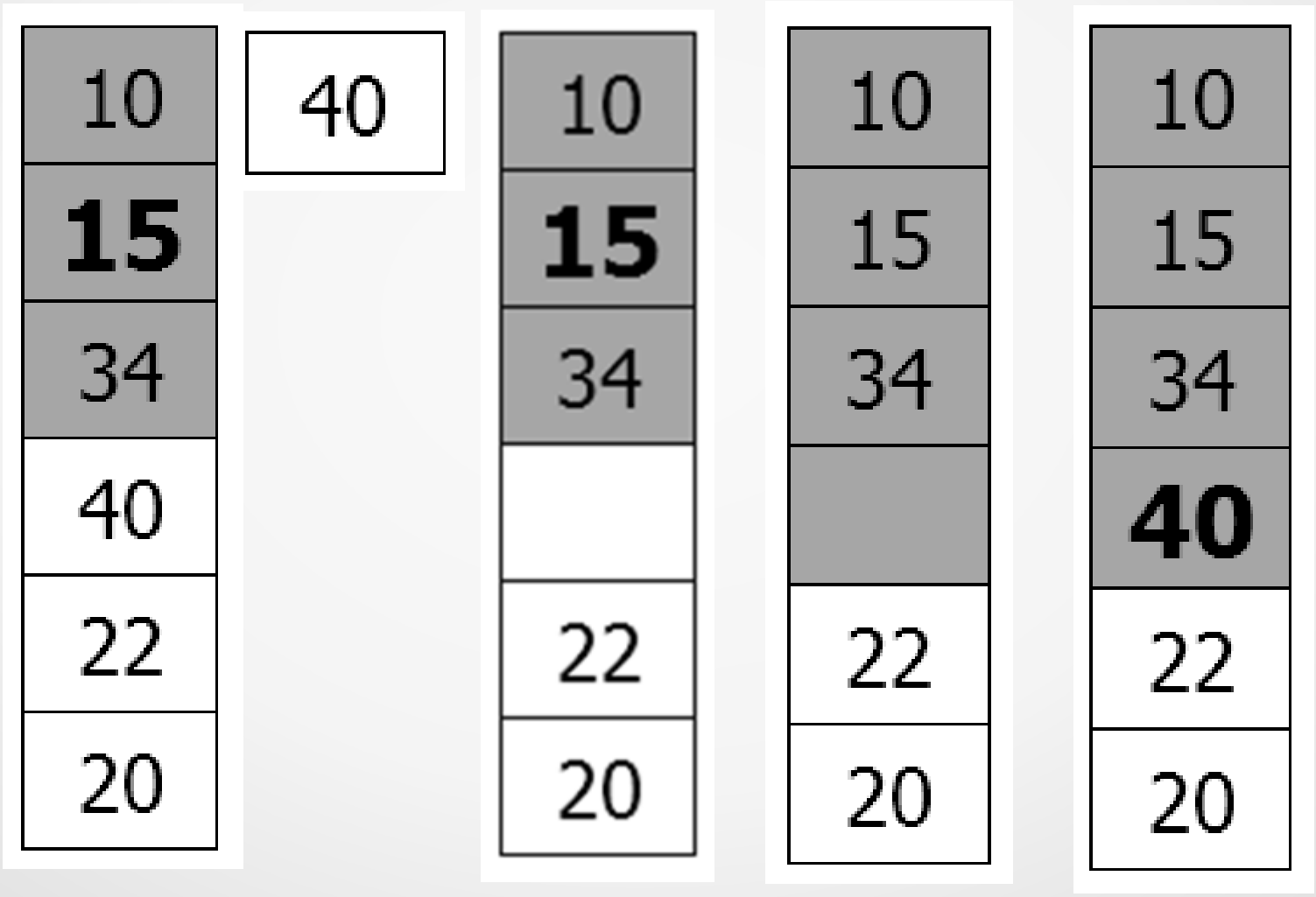
5.1.4 Inserción Simple



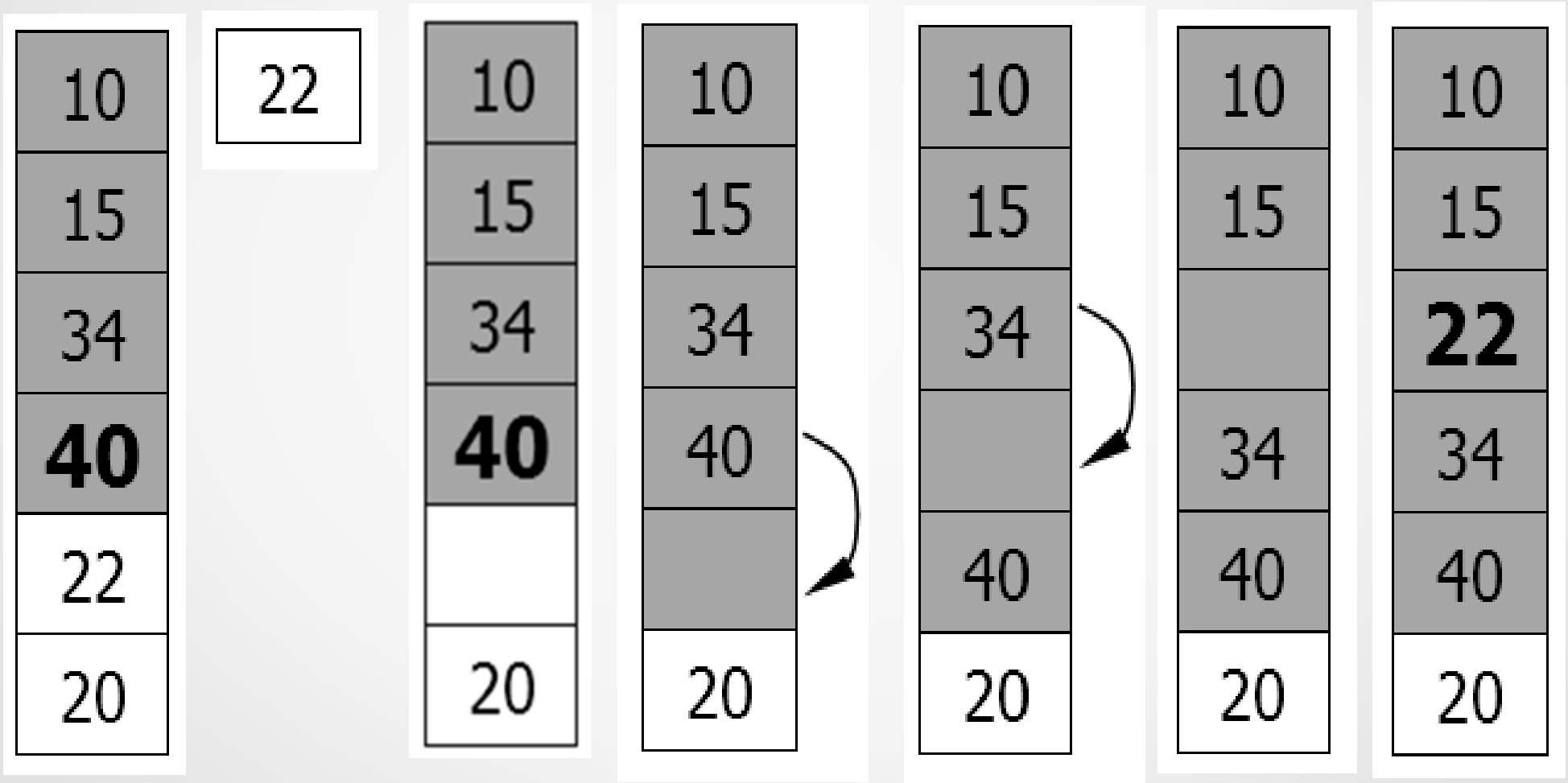
5.1.4 Inserción Simple



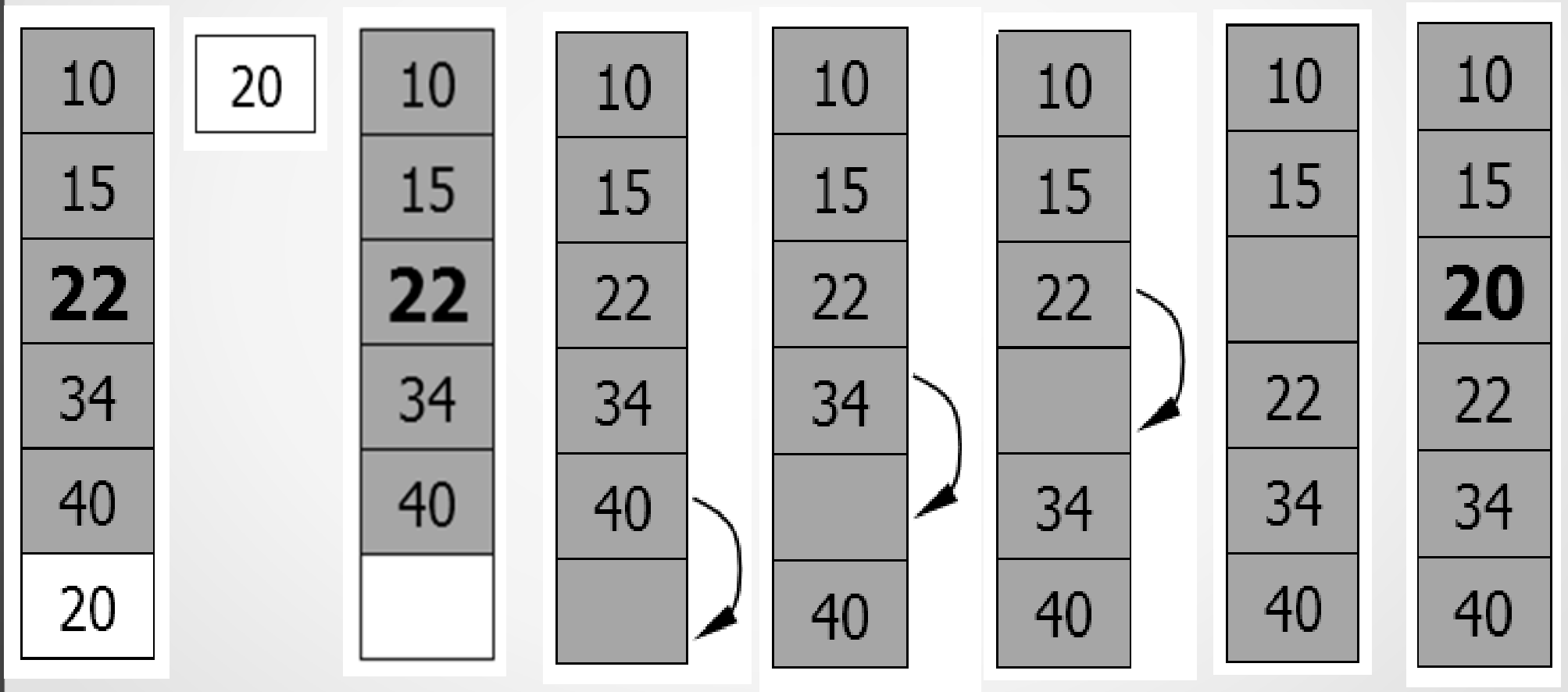
5.1.4 Inserción Simple



5.1.4 Inserción Simple



5.1.4 Inserción Simple



Código C++ para inserción simple (complételo para que se genere el vector al azar y que se cuente el numero de movimientos que se hacen).

```
for(int j=1;j<n;j++){
     int temp=archivo[j];
     int i=j-1;
     while ((archivo[i]>temp)and(i>=0)){
          archivo[i+1]=archivo[i];
          i--;
     }
     archivo[i+1]=temp;
}
```

5.1.5 Ordenamiento Shell

- No es un método de ordenamiento sino un refinamiento para otros.
- El Dr. Donald Shell sugirió que el archivo a ordenar **se divida** en forma lógica **en partes mas pequeñas** llamadas ***subarchivos*** (ya se sabía que **cuando los archivos son pequeños el esfuerzo para ordenarlos es menor**).
- El número y tamaño de los subarchivos se determina por un concepto llamado ***incremento:*** *distancia que hay entre 2 registros de un mismo subarchivo*.
- Se aplica una serie de incrementos en forma descendente pero el último debe ser 1.

5.1.5 Ordenamiento Shell

- Cada subarchivo se ordenará por separado de los demás.
- Se diseñó inicialmente para Inserción Simple ya que la cantidad de trabajo de ese método se debe a que las comparaciones se aplican a datos en direcciones consecutivas de memoria.

5.1.5 Ordenamiento Shell

Ejemplo:

44	55	12	42	94	18	6	67

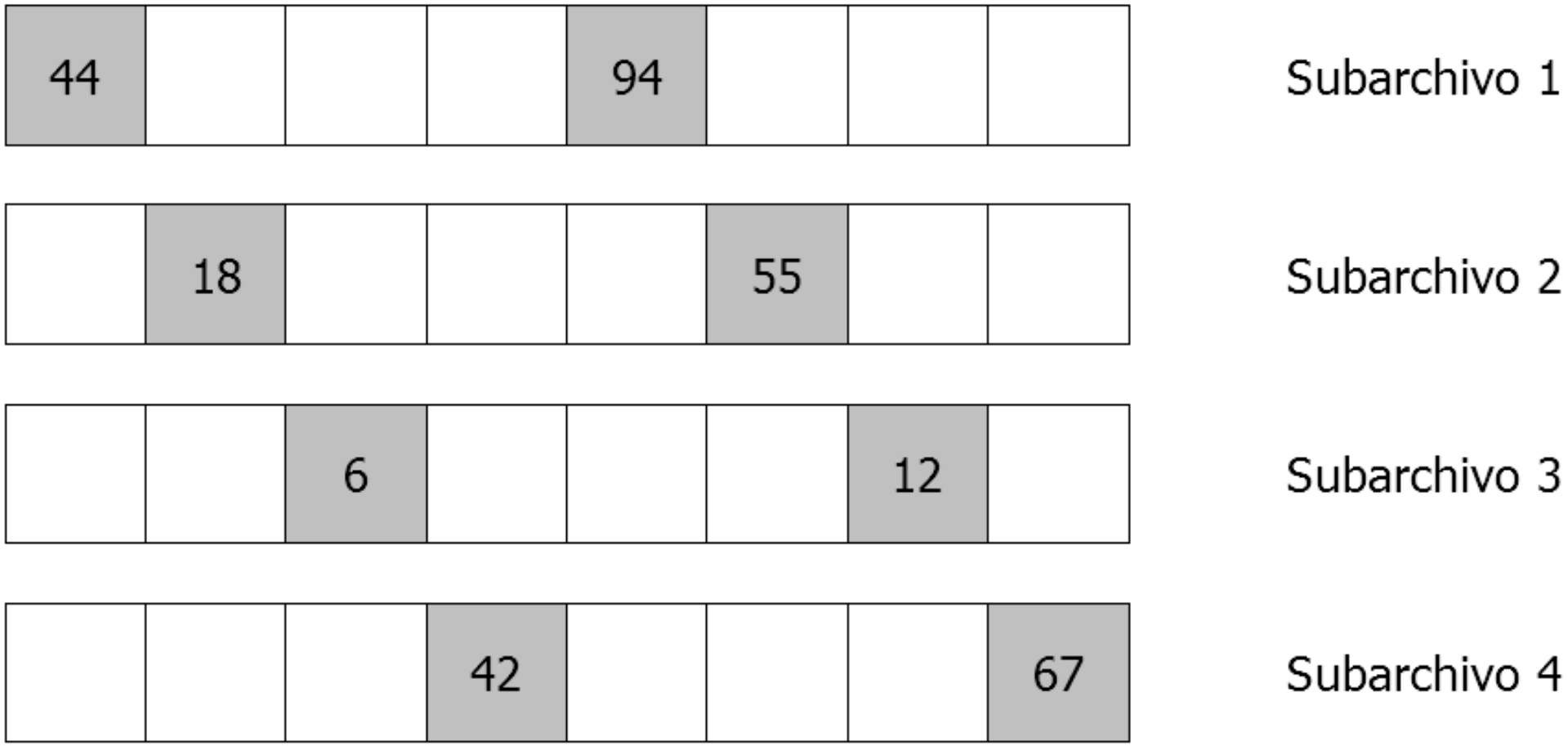
44	55	12	42	94	18	6	67

Supongamos que conviene aplicar un **incremento de 4** (el número de subarchivos que se obtiene es 4).

44				94				Subarchivo 1
	55				18			Subarchivo 2
		12				6		Subarchivo 3
			42				67	Subarchivo 4

5.1.5 Ordenamiento Shell

Los subarchivos se ordenan independientemente:



5.1.5 Ordenamiento Shell

El archivo antes de aplicado el **incremento 4:**

44	55	12	42	94	18	6	67

Una vez aplicado el incremento 4 y ordenados por separado:

44	18	6	42	94	55	12	67

El archivo no quedó ordenado, pero se ha demostrado, que los registros queden más cerca de su lugar definitivo.

5.1.5 Ordenamiento Shell

44	18	6	42	94	55	12	67

Se aplica el siguiente incremento, dos (2) por ejemplo.

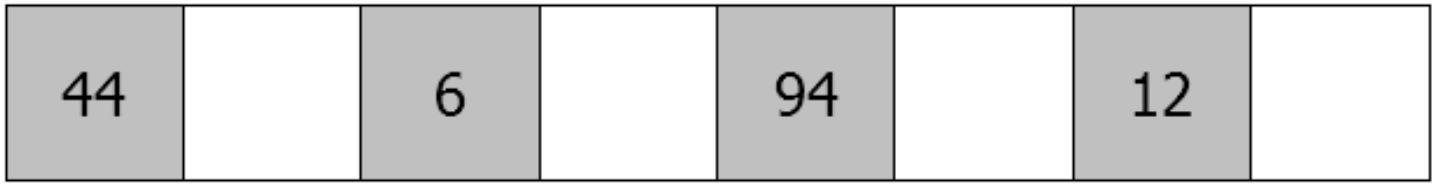
44		6		94		12	

Subarchivo 1

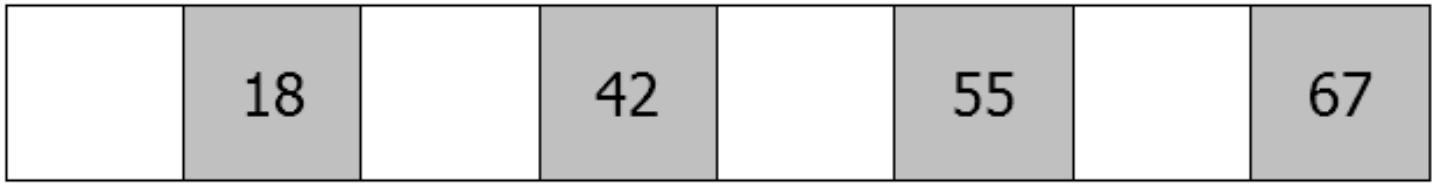
	18		42		55		67

Subarchivo 2

5.1.5 Ordenamiento Shell



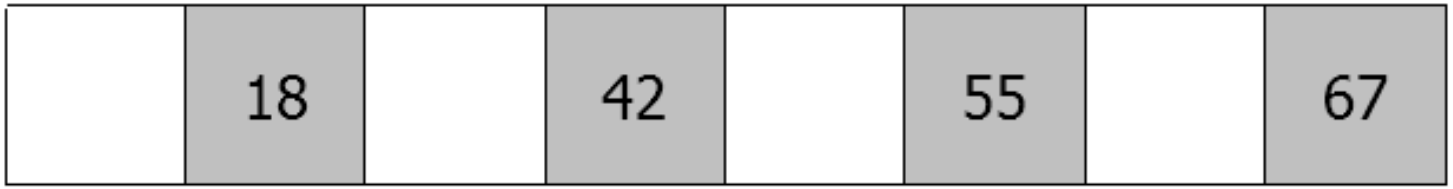
Subarchivo 1



Subarchivo 2

Se ordenan los subarchivos

Subarchivo 1



Subarchivo 2

5.1.5 Ordenamiento Shell

El archivo completo:

6	18	12	42	44	55	94	67

¿Cómo se encuentra el archivo?, ¿con las llaves en una mejor posición?

5.1.5 Ordenamiento Shell

Finalmente se debe usar incremento 1:

6	18	12	42	44	55	94	67

Subarchivo 1

Una vez ordenado el único subarchivo, el archivo completo quedará:

6	12	18	42	44	55	67	94

Comparación de Inserción Simple para un archivo completo y para un subarchivo (SHELL).

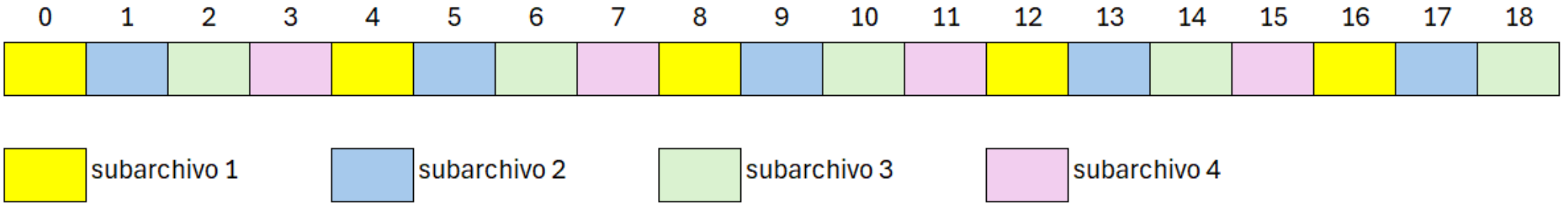
```
for(int j=1; j<n; j++){
    int temp=archivo[j];
    int i=j-1;
    while ((archivo[i]>temp)and(i>=0)){
        archivo[i+1]=archivo[i];
        i--;
    }
    archivo[i+1]=temp;
}
```

```
for(long j=subarchivo+increm-1; j<n; j=j+increm){
    long temp=archivo[j];
    long i=j-increm;
    while ((archivo[i]>temp)and(i>=subarchivo-1)){
        archivo[i+increm]=archivo[i];
        i = i - increm;
    }
    archivo[i+increm]=temp;
}
```

El subarchivo 1 empieza en la posición 0, el subarchivo 2 empieza en la posición 1 y así sucesivamente.

j toma como valor inicial el segundo elemento del subarchivo
(la dirección "increm" es el 2o elemento del subarchivo 1)

Ejemplo: **increm = 4**



Ejercicio

Escribir un programa para ordenar **uno** de los subarchivos de un vector, con base en el algoritmo básico del Dr. Shell:

- Genere al azar un archivo de cierto tamaño pequeño.
- Imprima en la pantalla el archivo recién generado.
- Solicite al usuario que indique que incremento se va a usar para ordenar uno de los subarchivos.
- Solicite al usuario que indique el subarchivo deseado.
- Aplique el algoritmo de la diapositiva anterior.
- Muestre el contenido del vector luego del proceso y compruebe que se haya realizado correctamente el ordenamiento del subarchivo indicado. El resto de las llaves deben quedar sin cambio.

5.1.5 Ordenamiento Shell

Análisis del Algoritmo Shell

- Los archivos pequeños requieren menos trabajo para ordenarse.
- La **descomposición lógica** de un archivo en segmentos más pequeños, ocasiona que el esfuerzo total sea menor.
- Los movimientos de los datos cuando el incremento es grande, acercan los datos a su lugar definitivo.
- La secuencia de incrementos podría ser seleccionada arbitrariamente.
- Única recomendación: los incrementos no deben ser múltiplos unos de otros.

5.1.5 Ordenamiento Shell

Donald Knuth recomienda las dos secuencias de incrementos siguientes (escritas en ORDEN INVERSO):

1, 4, 13, 40, 121, ... donde $\mathbf{H_{i-1} = 3H_i + 1}$

y

1, 3, 7, 15, 31, ... donde $\mathbf{H_{i-1} = 2H_i + 1}$

Empíricamente se ha comprobado que el orden de complejidad temporal del algoritmo Shell es $\mathbf{n^{1.2}}$ (enseguida se muestra una tabla de demostración). El código C++ se puede descargar del sitio www.felipealanis.org

Ordenamiento SHELL Completo

```
#include <iostream>
#include "time.h"
#include "stdio.h"
#include "stdlib.h"
using namespace std;
long long contador=0;
void OrdenaSubarchivo(long,long);
long const n=100000;
long archivo[n];
int main()
{
    srand(time(NULL));
    for(long i=0;i<n;i++) {
            archivo[i]=rand()%(n*10)+1;
    }
    if (n<20){
            cout << "Vector original" << endl;
            for(long i=0;i<n;i++)
                    cout << archivo[i] << " ";
            cout << endl;
    }
    // ------------ Inicia Shell con Inserción Simple --------------
    long incremento = long(n/2)-1;
    while (incremento>1){
            cout << "incremento usado = " << incremento << endl;
            for(long subarchivo=1;subarchivo<=incremento;subarchivo++)
                    OrdenaSubarchivo(incremento,subarchivo);
            incremento = long(incremento/2)-1;
    }
    cout << "al final se usa obligatoriamente incremento = 1" << endl;
    OrdenaSubarchivo(1,1);
    // ------------- Fin Shell con Inserción Simple ------------------------
    if (n<20){
            cout << "Vector ordenado" << endl;
            for(long i=0;i<n;i++)
                    cout << archivo[i] << " ";
            cout << endl;
    }
    cout << "Tamanio del archivo    : " << n << endl;
    cout << "Numero de corrimientos : " << contador << endl;
    system("pause");
    return 0;
}

void OrdenaSubarchivo(long increm,long subarch){
    // el subarchivo 1 empieza en la posición 0, el subarchivo 2 empieza en la posición 1 y asi sucesivamente
    // j toma como valor inicial empieza en el segundo elemento del subarchivo (la dirección "increm" es el 2o elemento del subarchivo 1)
    for(long j=subarch+increm-1;j<n;j=j+increm){
            long temp=archivo[j];
            long i=j-increm;
            while ((archivo[i]>temp)and(i>=subarch-1)){
                    contador++;
                    archivo[i+increm]=archivo[i];
                    i = i - increm;
            }
            archivo[i+increm]=temp;
    }
}
```

5.1.5 Ordenamiento Shell

Cantidad de trabajo realizado por ShellSort y comprobación del orden de complejidad temporal $n^{1.2}$

n	Número de movimientos	$n^{1.2}$	movimientos / $n^{1.2}$
10,000	106,039	63,095.73	1.6806
50,000	744,459	435,275.28	1.7103
100,000	1,737,630	1,000,000.00	1.7376
200,000	3,824,850	2,297,396.71	1.6649
250,000	5,247,470	3,002,811.08	1.7475
300,000	5,991,380	3,737,192.82	1.6032

5.2 Métodos Externos de Ordenamiento

Los métodos externos de ordenamiento requieren de una cantidad de espacio extra proporcional al tamaño del archivo.
Sin embargo, no necesariamente se requiere espacio para todos los registros; dependiendo del método y del algoritmo empleado, solo se requerirá espacio extra para las llaves o para apuntadores a los registros.

5.2.1 Mezclas

Este grupo de algoritmos toma 2 archivos ordenados y produce un nuevo archivo también ordenado.

5.2.1.1 Mezcla Simple

arch1 y **arch2** son dos archivos ya ordenados.

Por ejemplo:

arch1	5	10	28	31	55	92	100	131	139

arch2	3	7	8	40	56

arch3 deberá contener todos los registros de arch1 y arch2 también en orden.

Algoritmo para Mezcla Simple

1. Leer el primer registro de cada archivo (**arch1** y **arch2**)
2. Crear un archivo nuevo vacío y llamarlo **arch3**.
3. Se comparan las llaves de **arch1** y **arch2**.
   - El registro con la llave menor de los dos se guardará en **arch3**.
   - Se toma el siguiente registro de **arch1** o **arch2** (según de donde se haya tomado el último que se escribió en **arch3**).
   - Ir al paso 3.
4. Cuando uno de los archivos llegue al final, los registros restantes del otro se traspasarán a **arch3**.

Ejemplo de ejecución del algoritmo

arch1	5	10	28	31	55	92	100	131	139
arch2	3	7	8	40	56				

arch3	3												
	3	5											
	3	5	7										
	3	5	7	8									
	3	5	7	8	10								
	3	5	7	8	10	28							
	3	5	7	8	10	28	31						
	3	5	7	8	10	28	31	40					
	3	5	7	8	10	28	31	40	55				
	3	5	7	8	10	28	31	40	55	56			
	3	5	7	8	10	28	31	40	55	56	92	100	131	139
arch3	3	5	7	8	10	28	31	40	55	56	92	100	131	139

Ventajas:

- Orden de complejidad **temporal O(n)**

Desventajas:

- Se requiere una cantidad de espacio extra equivalente a la unión de ambos archivos.
- Este método de ordenamiento es viable solo si los archivos origen están previamente ordenados.

5.2.1.2 Mezcla Directa

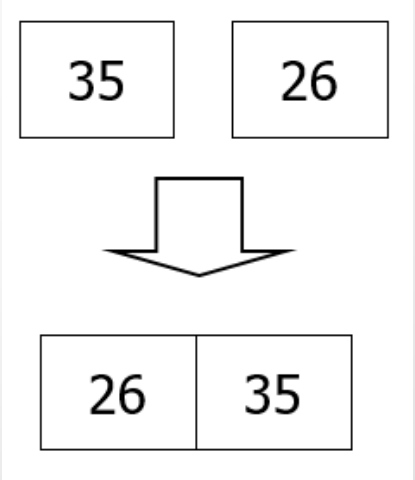
Fundamentos:

- De acuerdo con el algoritmo de Mezcla Simple sabemos que se puede obtener un archivo ordenado a partir de dos previamente ordenados.
- Los archivos de tamaño **UNO** ya están ordenados.

Dado el siguiente archivo al azar:

35	26	80	94	10	15	21	3	8	18	68	42	51

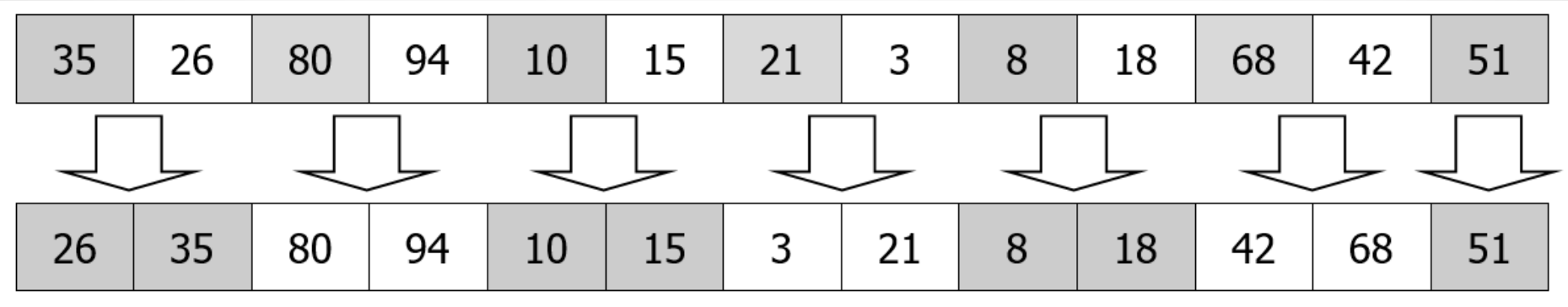
Se toman las 2 primeras llaves del archivo y se sugiere se vean como 2 archivos independientes **de tamaño 1 (al ser de tamaño 1, ya se encuentran ordenados)**, si los mezclamos usando el algoritmo se mezcla simple para formar un solo archivo, el resultado sería:



5.2.1.2 Mezcla Directa

Primera Etapa:

EL algoritmo de mezcla directa nos indica que inicialmente hagamos mezclas simples en parejas de archivos de tamaño 1:

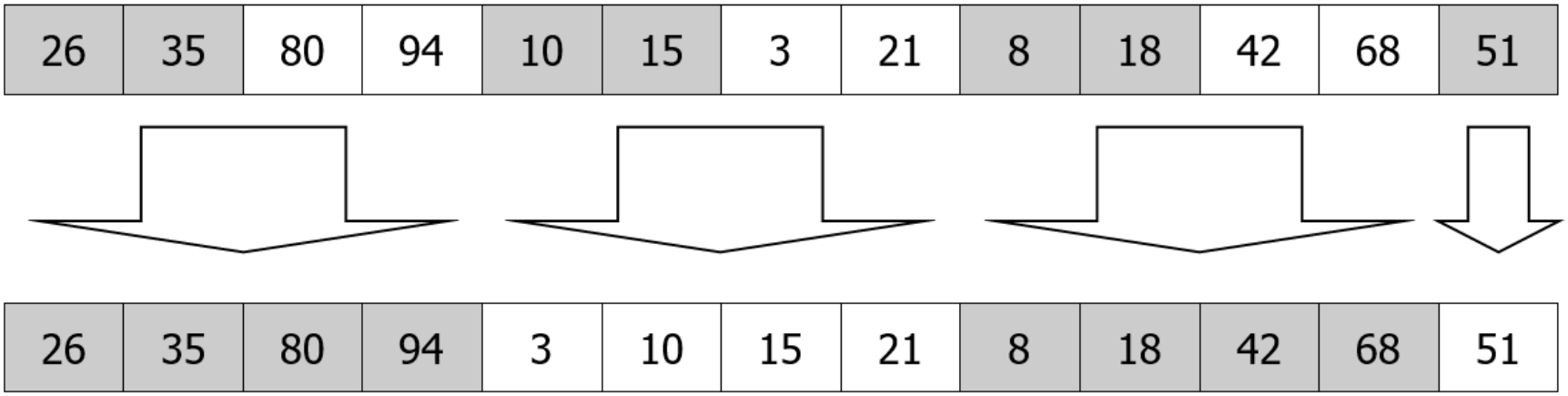


Observe que al ser impar el tamaño del archivo, la última llave no se mezcla con ninguna otra.

5.2.1.2 Mezcla Directa

Segunda Etapa:

Luego de la primera etapa, el archivo se conforma de parejas de llaves ordenadas, por lo que se pueden hacer mezclas simples de nuevo.

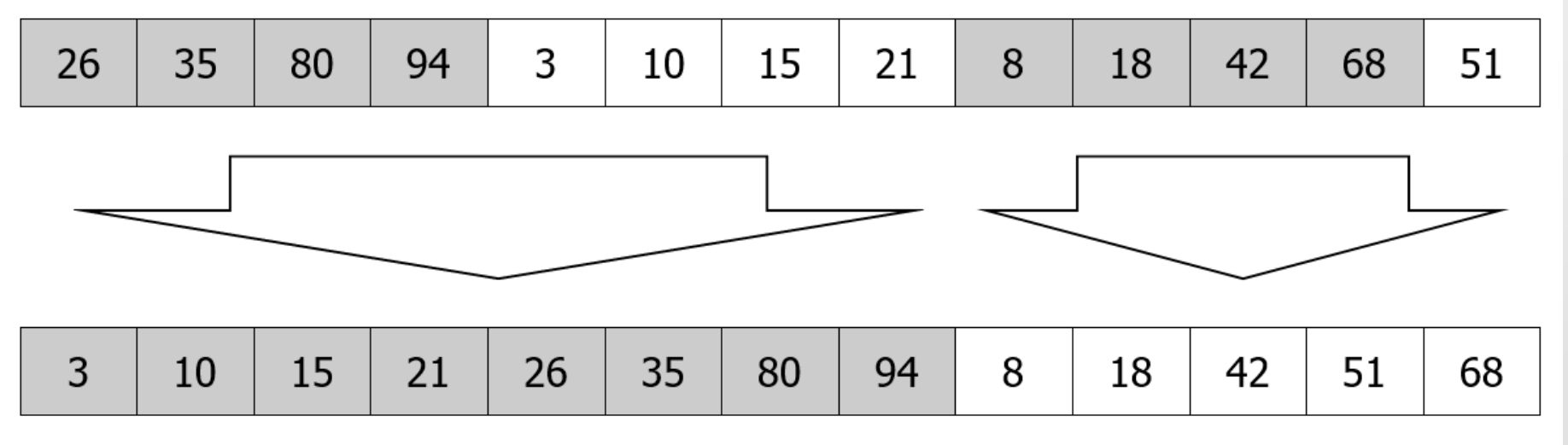


Observe que la última llave sigue sin mezclarse.

5.2.1.2 Mezcla Directa

Tercera Etapa:

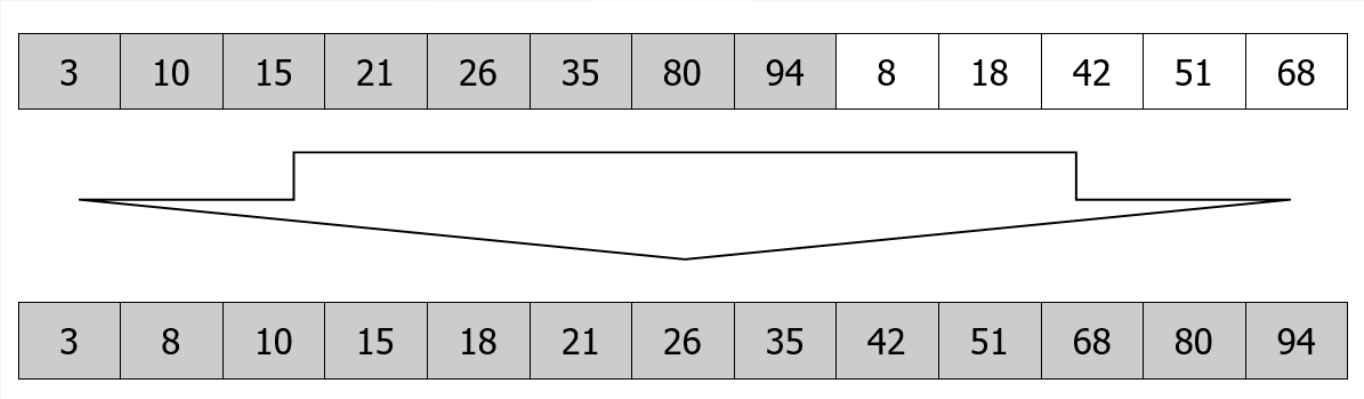
El archivo ahora se conforma de grupos ordenados de 4 llaves c/u, por lo que se puede hacer una mezcla de nuevo. Observe que hay un grupo de 4 llaves que se mezclará con la llave que no se había mezclado en las primeras dos etapas.



5.2.1.2 Mezcla Directa

Cuarta Etapa:

El archivo ahora se conforma de grupos ordenados de 8 llaves c/u (aunque, en este caso, al ser **N** pequeño, el otro único grupo ordenado es de solo 5 llaves). Se realiza la **mezcla final** entre los **dos subarchivos**.



Al aplicar sucesivamente mezclas simples a grupos de llaves, se puede lograr obtener ordenado un archivo cualquiera.

Los grupos se forman sucesivamente de tamaño $\mathbf{2^m}$, siendo m de un valor mínimo de **0** y máximo tal que $\mathbf{2^m}$ nunca sea mayor que el tamaño del archivo (**N**).

El orden de complejidad temporal de este algoritmo **O(n log n)**.

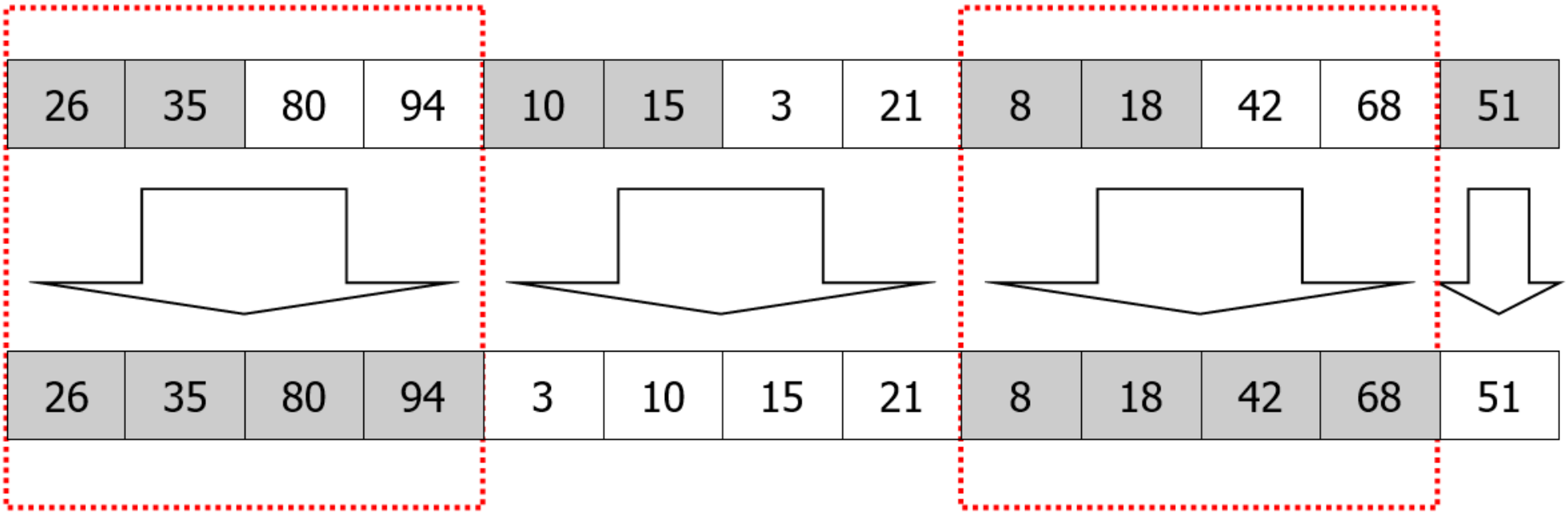
Ejercicio

Escriba un programa en C++ para ordenar un vector con valores numéricos generados al azar usando **mezcla directa** o pida a una de las herramientas de inteligencia artificial que lo haga por usted pidiéndole que le explique paso a paso las diferentes procesos realizados.

5.2.1.3 Mezcla Natural

El archivo original que se usó para el ejemplo de Mezcla Directa es aleatorio, sin embargo, se observa como en la segunda etapa, la **primera** y la **tercera parejas de subarchivos** a mezclar, ya conforman de por sí subarchivos **ordenados**.

Esto significa que el algoritmo de mezcla directa puede hacer trabajo innecesario. La mezcla natural resuelve este problema potencial.



5.2.1.3 Mezcla Natural

Se trabajará con el mismo archivo que se usó para el ejemplo de mezcla directa:

35	26	80	94	10	15	21	3	8	18	68	42	51

En la figura siguiente se muestran los subarchivos lógicos de acuerdo a dos conceptos que en inglés se denominan ***runs*** y ***stepdowns***. Aquí le llamaremos **subidas** y **bajadas**.

35	26	80	94	10	15	21	3	8	18	68	42	51

Las **subidas** son los **subarchivos ordenados** y las **bajadas** son el final de una subida y el inicio de otra (cuando la siguiente llave, en vez de ser mayor, es menor).

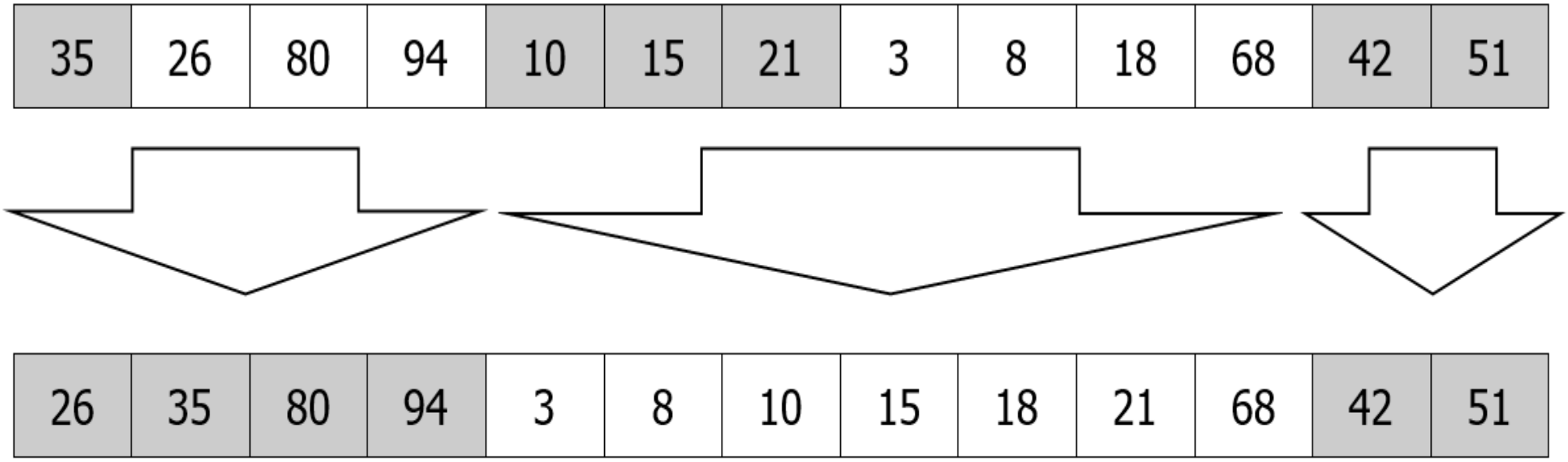
El número máximo de subidas que puede haber es **n**, y de bajadas, **n-1** (en caso de que el archivo esté ordenado a la inversa).

Observe que, si un archivo ya se encuentra ordenado, solo habrá una subida y ninguna bajada.

5.2.1.3 Mezcla Natural

Procedimiento de Mezcla Natural

Primera Etapa:



5.2.1.3 Mezcla Natural

Segunda Etapa:

26	35	80	94	3	8	10	15	18	21	68	42	51

3	8	10	15	18	21	26	35	68	80	94	42	51

5.2.1.3 Mezcla Natural

Tercera (y última) Etapa:

3	8	10	15	18	21	26	35	68	80	94	42	51

3	8	10	15	18	21	26	35	42	51	68	80	94

Como se ve en el ejemplo, se requieren menos etapas usando Mezcla Natural que con Mezcla Directa, sin embargo, el orden de complejidad temporal también es **O(n log n)** ya que el peor caso es cuando está ordenado a la inversa y se comporta como mezcla directa.

Ejercicio

Escriba un programa en C++ para ordenar un vector con valores numéricos generados al azar usando **mezcla natural** o pida a una de las herramientas de inteligencia artificial que lo haga por usted pidiéndole que le explique paso a paso las diferentes procesos realizados.

5.2.2 Distribución Simple

Métodos de Ordenamiento por Distribución(por Paquetes)

ANALOGÍA

Una persona tiene que ordenar alfabéticamente (por apellidos) un paquete de solicitudes de ingreso de estudiantes a una escuela, el número de solicitudes es grande.
Una vez que el encargado de organizar las hojas las recibe, puede decidir entre:

1. Realizar el ordenamiento considerando a todo el paquete en su conjunto.
2. Tomar una a una las hojas de solicitud y acomodarlas para formar 26 paquetes de solicitudes independientes (uno para cada letra del alfabeto). Al terminar este procedimiento, los ordenará en forma independiente, finalmente solo restará unir los paquetes ya ordenados.

5.2.2 Distribución Simple

La segunda opción de la página anterior es un método de los llamados ***ordenamiento por distribución***.

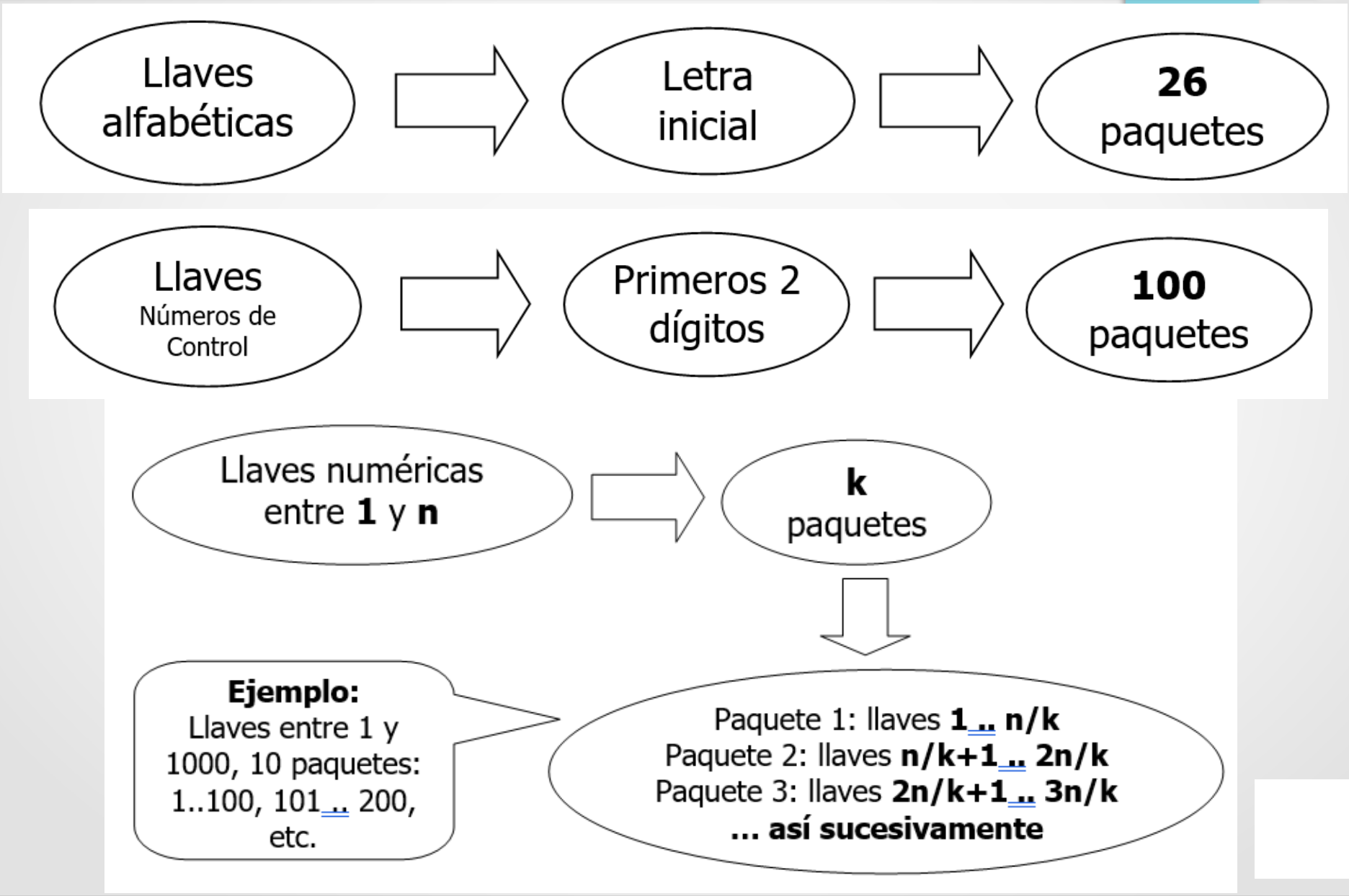
Este tipo de métodos se componen de tres fases:

Distribución en paquetes.
Ordenamiento de los paquetes.
Combinación de los paquetes.

La combinación debe ser un proceso simple (**unión**).

Hay que conocer bien las llaves para elegir como hacer la distribución:

5.2.2 Distribución Simple

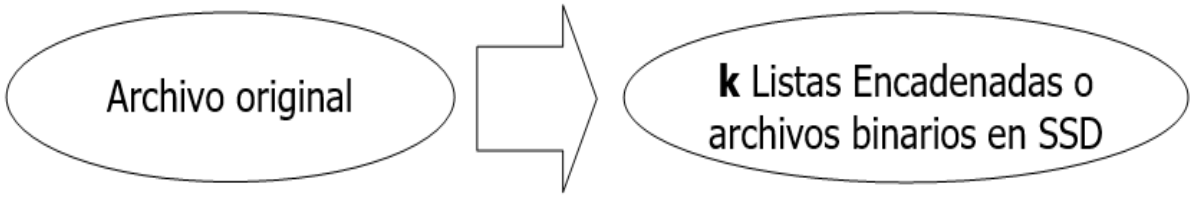


5.2.2 Distribución Simple

Etapas de los métodos de Distribución:

- **Distribución**
- **Ordenamiento**
- **Combinación**

Etapa 1. Distribución



Etapa 2. Ordenamiento

Se puede ordenar el paquete mientras se va formando, evitando la manipulación doble de nodos

Por lo que se fusionan las 2 primeras etapas.

Etapa 3. Combinación

Se toman los registros de cada paquete, uno a uno y se escriben en un archivo con el mismo nombre que el original o en un nuevo archivo. Una vez terminado este proceso, el archivo resultante estará ordenado.

5.2.2 Distribución Simple

Análisis

Distribuir en paquetes ocasiona un ahorro en esfuerzo

Los archivos pequeños se ordenan con menos esfuerzo que los grandes.

- El orden de complejidad temporal es **$O(n^2)$** para una lista lineal ordenada.

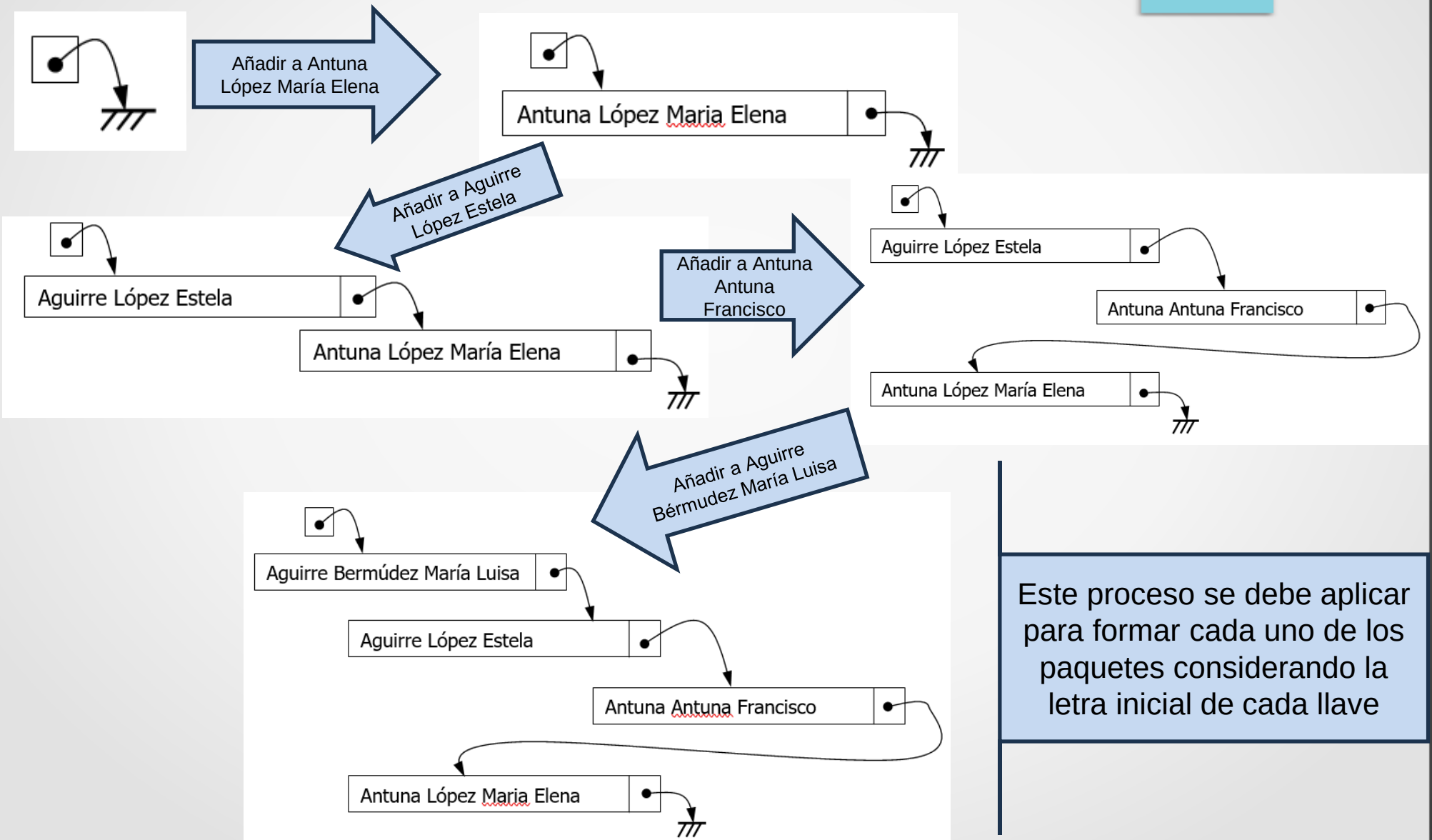
1 archivo de **1000** registros digamos **1'000,000** instrucciones ejecutadas
1 archivo de 1000 registros dividido en 26 paquetes:

- número de registros x paquete ≈ **40**
- instrucciones ejecutadas x paquete ≈ **1,600**
- instrucciones acumuladas = 1600 x 26 ≈ **40,000**

Aunque el orden de complejidad temporal es **$O(n^2)$**,
Distribución Simple ocasiona disminución de trabajo.

Se llama Distribución Simple porque solo hay **una distribución ...**
luego el ordenamiento de los paquetes ....
y finalmente **una combinación**.

Ejemplo de la formación del paquete de la letra **"A"**



Este proceso se debe aplicar para formar cada uno de los paquetes considerando la letra inicial de cada llave

Ejercicio 1

- Escribir un programa que cree un archivo que contenga llaves alfabéticas al azar (apellidos y nombre de personas).
- Los apellidos y/o nombres de las personas pueden contener cualquier carácter de la tabla ASCII, pero deben iniciar con una letra del alfabeto inglés, es decir, no podrán iniciar con **eñe**, a menos que usted considere esa posibilidad al manipular los paquetes (ejercicio 3).

Un archivo .TXT es una buena opción ya que Distribución Simple se ideó para archivos de acceso secuencial

Ejercicio 2

- Tomar una a una las llaves del archivo del Ejercicio 1 (supongamos que se llama *Datos.txt*) y usando **una** lista encadenada sencilla, ordenarlas y guardarlas ya ordenadas en un archivo con un nombre temporal.
- Al final se debe renombrar el archivo original (*Antiguo.txt* por ejemplo) para que el archivo nuevo ya ordenado sea renombrado a *Datos.txt*. De esta manera se conservan los dos archivos por cualquier eventualidad.

Este ejercicio no aplica Distribución ya que solo se usa una lista encadenada.

Ejercicio 3

- Usando un **vector** de **listas encadenadas**, realizar el ordenamiento del archivo usando Distribución Simple.
- Se usarán, 26 listas encadenadas, una para cada paquete, de acuerdo a la letra inicial del apellido.
- Al final de la distribución y ordenamiento, el contenido de las listas debe ser vaciado a un archivo con el mismo nombre que el original (este se renombrará).
- Los índices del arreglo deben ser de 0→25 correspondientes a las listas de la 'A' → 'Z').

Este ejercicio aplica Distribución Simple para disminuir la cantidad de trabajo al ordenar el archivo

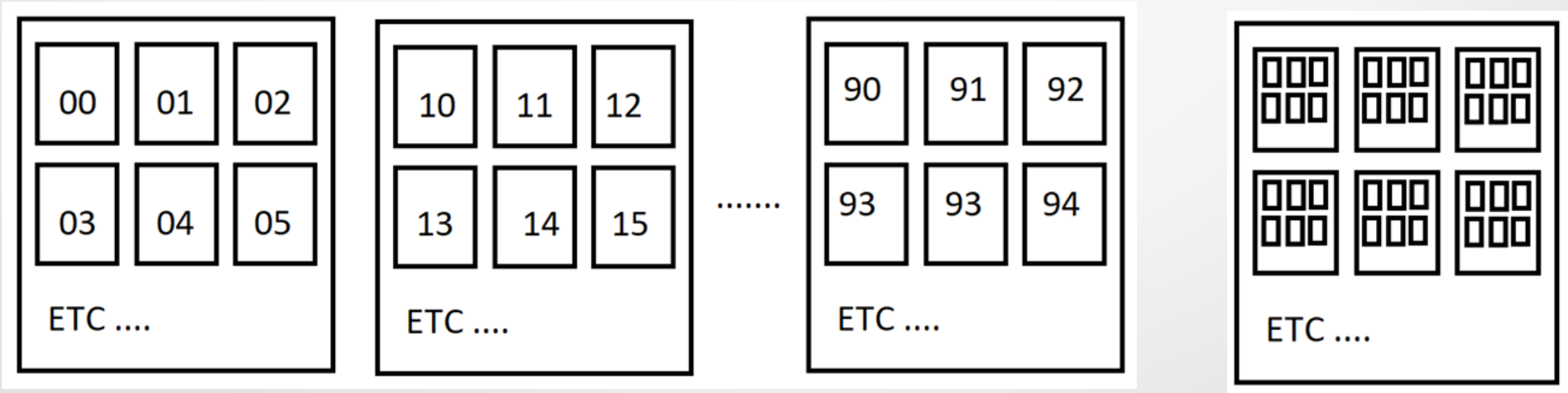
5.2.3 RADIX

Tomemos un archivo con llaves que son **cadenas de 5 dígitos** como el siguiente:

39216	43218	12354	84791	31821	42118	38216	13354	54161	01547	05890

El siguiente razonamiento logrará el ordenamiento

- Distribuir en 10 paquetes de acuerdo con el dígito **más** significativo (como si fuera distribución simple).
- No ordenar cada paquete
  - En vez de ello, mientras se hace la primera distribución, hacer otra en base al 2º dígito,
  - Mientras se hace la 2a distribución hacer una más en base al 3er dígito.
  - Lo mismo hasta el último dígito.



5.2.3 RADIX

Para resolver mediante el **razonamiento anterior**, se requiere un programa recursivo con una necesidad importante de espacio extra.

- Los primeros 10 paquetes no pueden ser formados hasta en tanto no se formen los siguientes 100.
- Y los 100 no se formarán hasta que no se formen los siguientes 1,000
- Y así sucesivamente.
- Ninguna combinación podrá efectuarse hasta en tanto se realice la última distribución: **(100,000 paquetes)**

Radix es una solución inspirada en el razonamiento anterior:

- El tamaño de la llave determina el numero de distribuciones y combinaciones a realizar.
- Se inicia con el **dígito menos significativo** y termina con el más significativo.
- Los paquetes serán combinados después de cada distribución
- La necesidad de **comparar** las llaves desaparece.
- **El orden** obtenido en cada etapa **se debe conservar** para el inicio de la siguiente (como en una **COLA**).

5.2.3 RADIX

Análisis:

- De igual forma que Distribución Simple, el número de paquetes depende de la naturaleza de la llave.
- Si hay memoria RAM suficiente, conviene usar **Listas Encadenadas** con un **apuntador al final** de la Lista, para que se conserve el orden (como una cola).
- Además, con las L.E. podremos cambiar el orden de los datos **sin moverlos** en realidad.

Implementación:

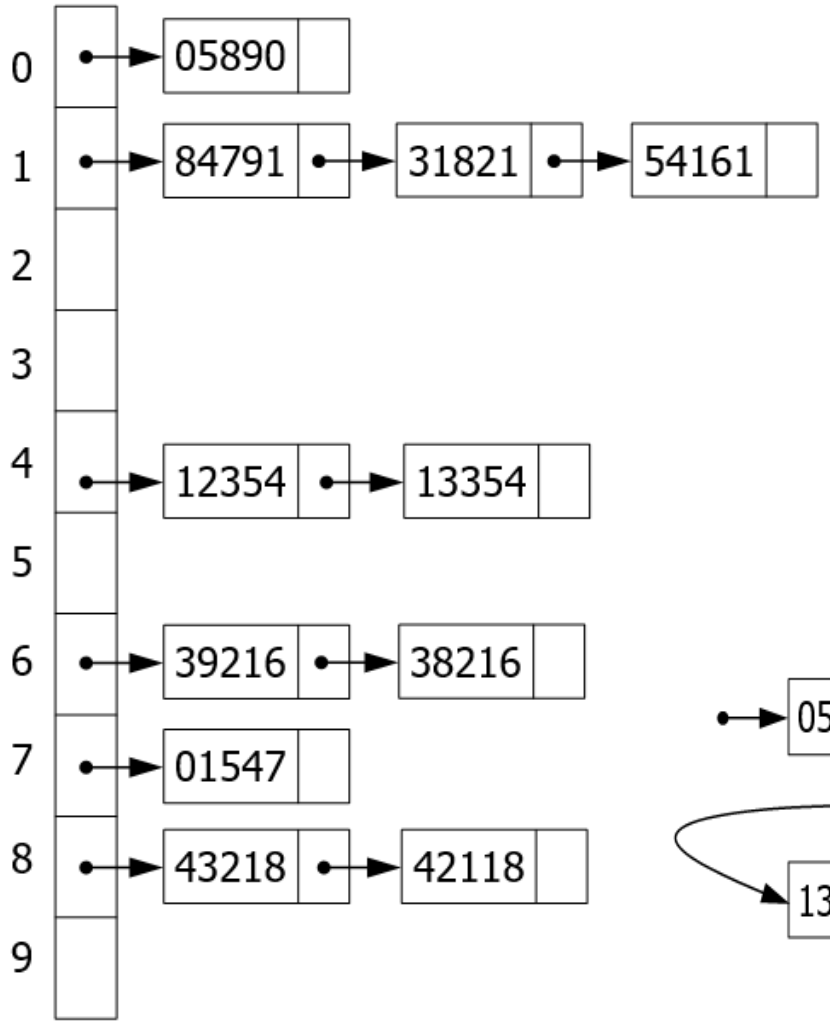
- Se requiere una Lista Encadenada para las Combinaciones además de las 10 listas para cada uno de los paquetes.

Ejemplo:

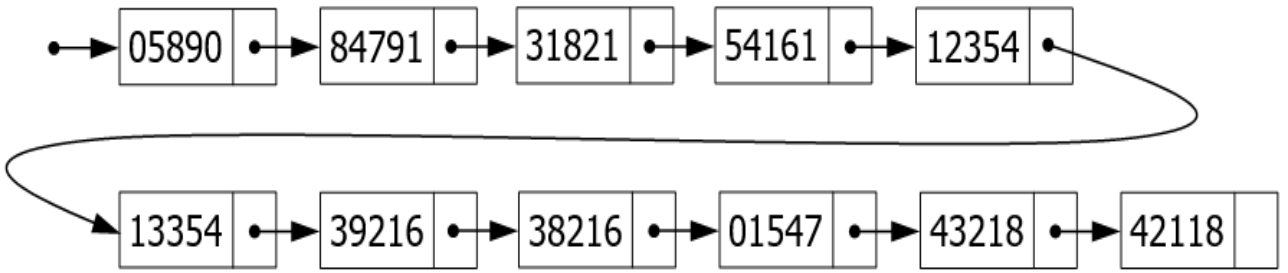
39216	43218	12354	84791	31821	42118	38216	13354	54161	01547	05890

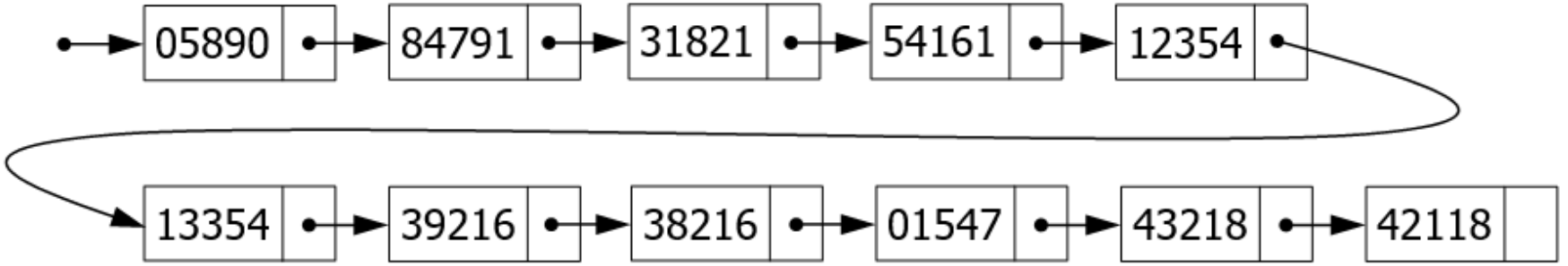
Primera Distribución (Dígito de la Posición 5):

(no se muestra el arreglo de apuntadores al último nodo para simplificar)

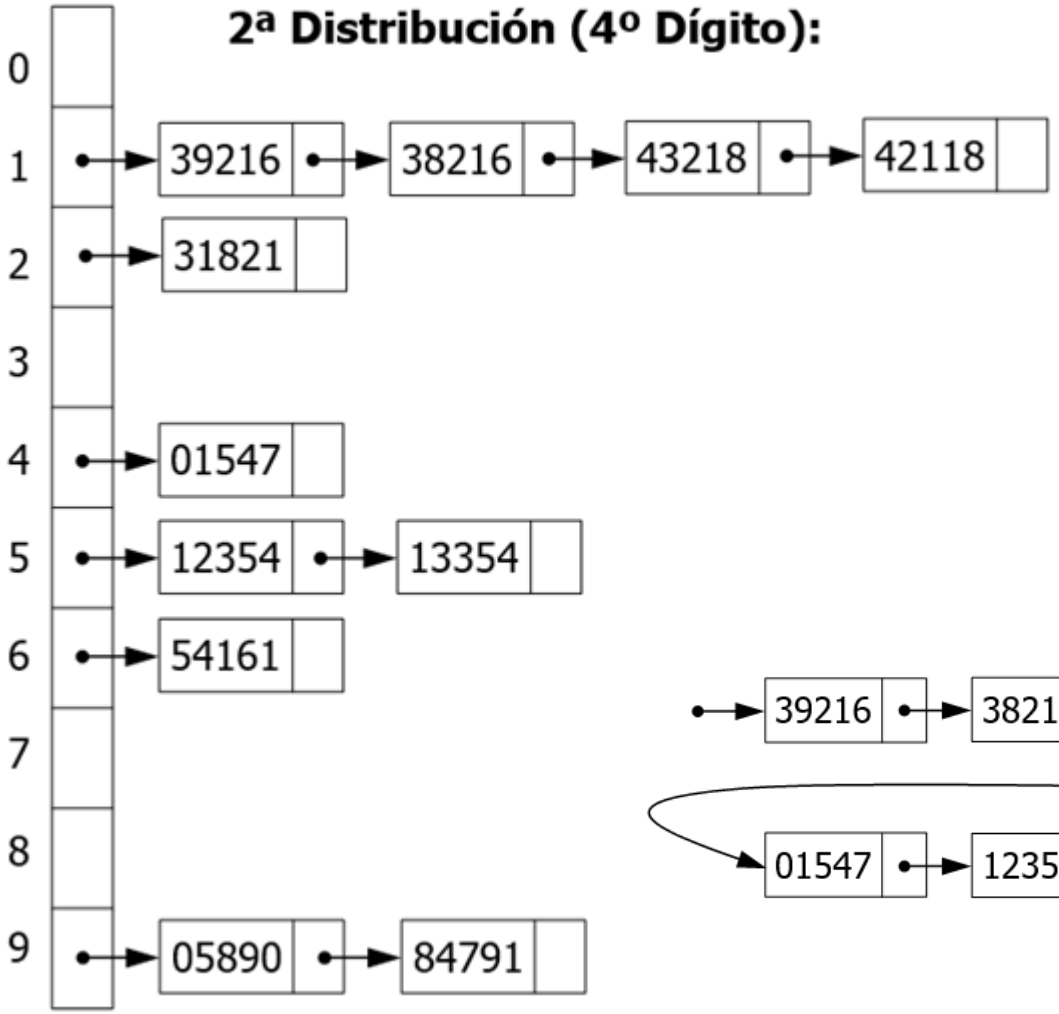


1ª Combinación

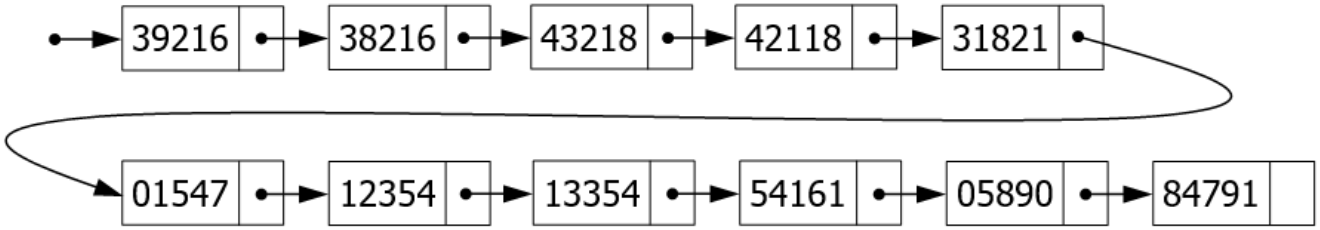


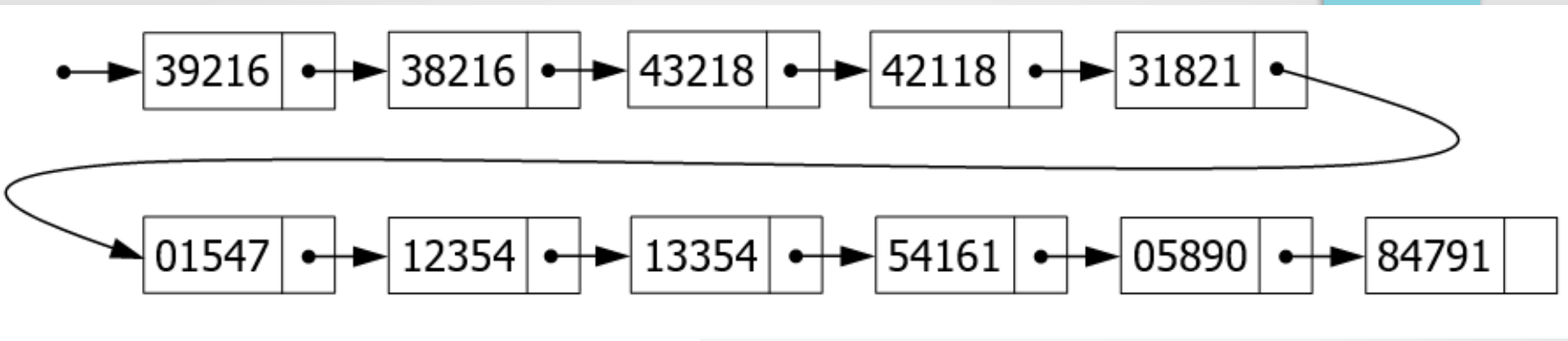


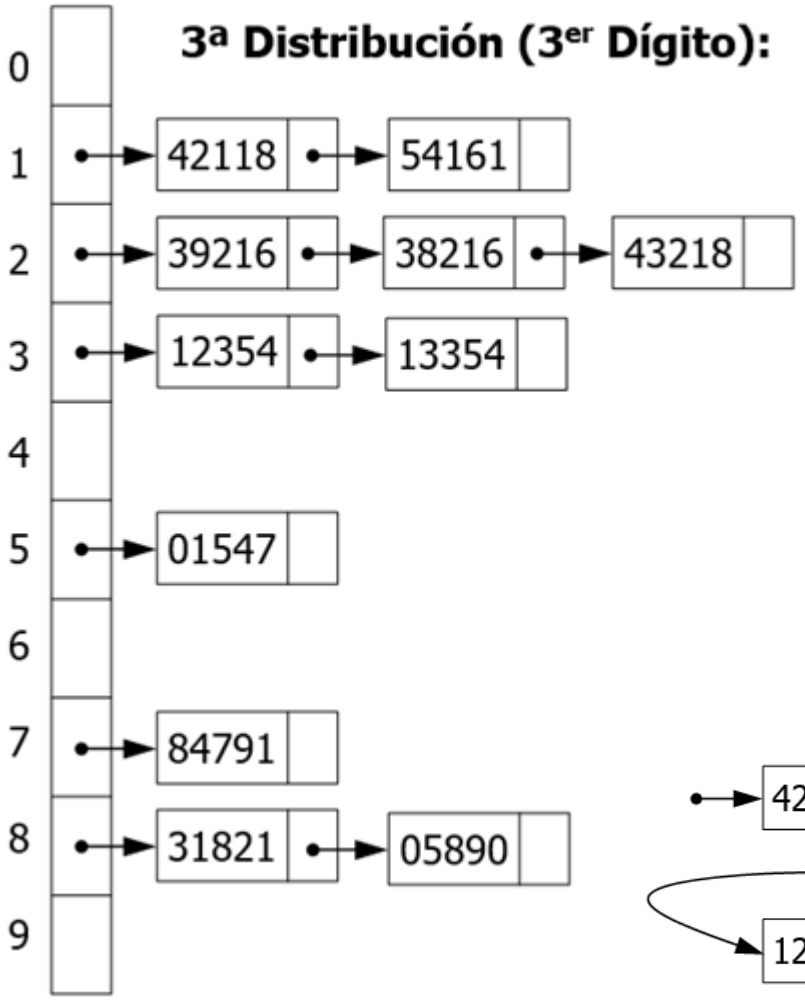
2ª Distribución (4º Dígito):

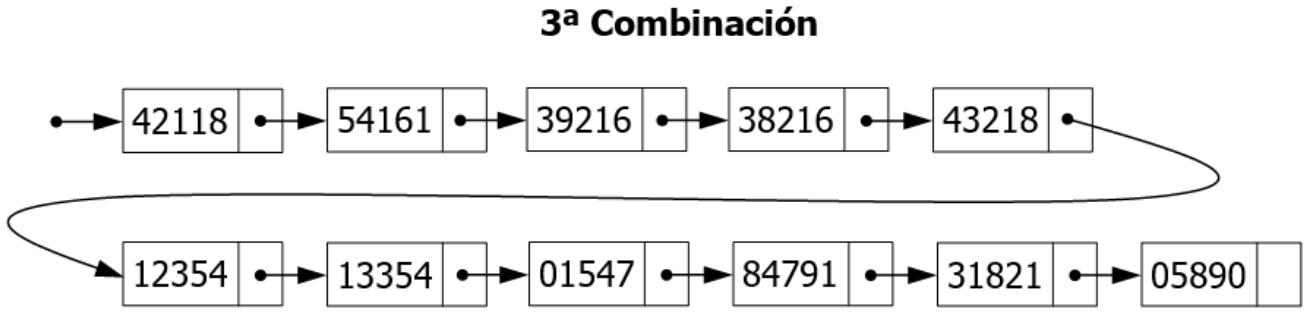


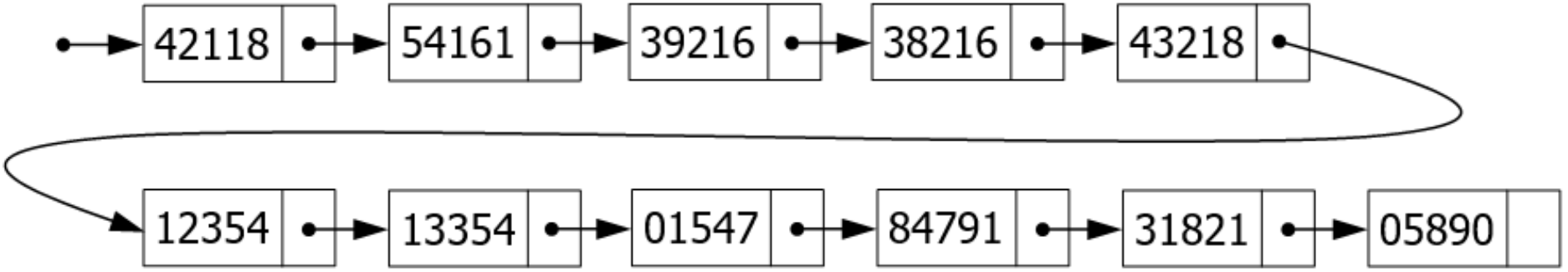
2ª Combinación



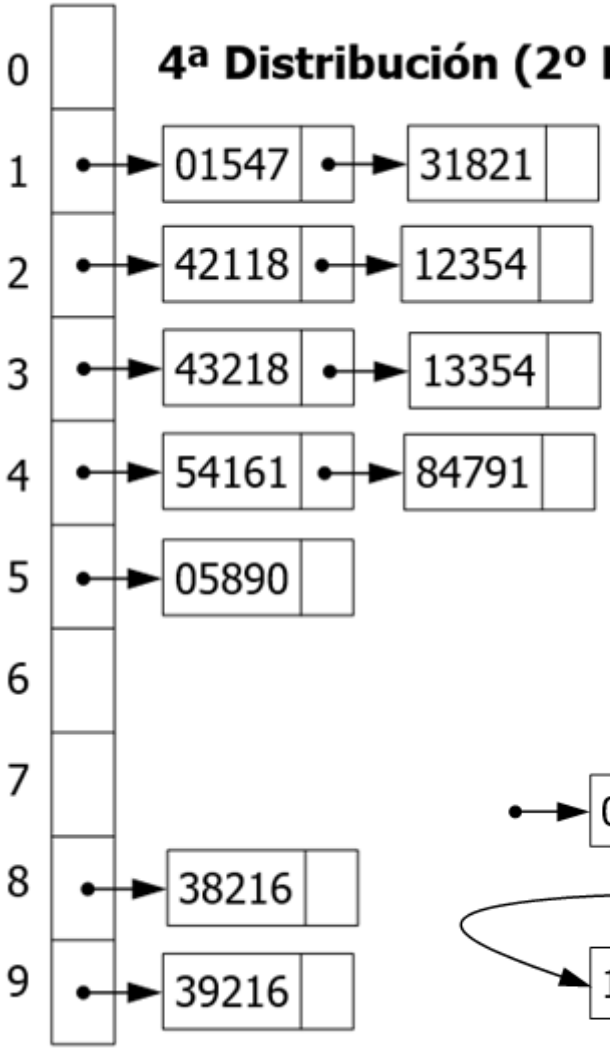

39216
38216
43218
42118
31821
01547
12354
13354
54161
05890
84791


3ª Distribución (3er Dígito):
0
1
42118
54161
2
39216
38216
43218
3
12354
13354
4
5
01547
6
7
84791
8
31821
05890
9

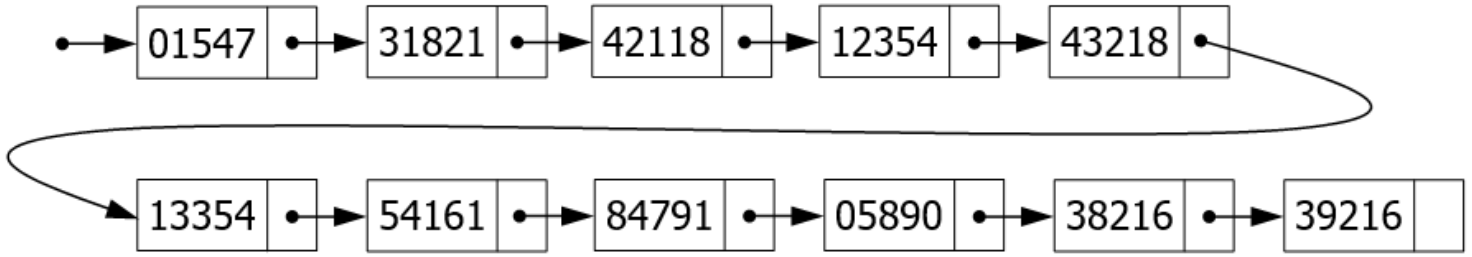

3ª Combinación
42118
54161
39216
38216
43218
12354
13354
01547
84791
31821
05890

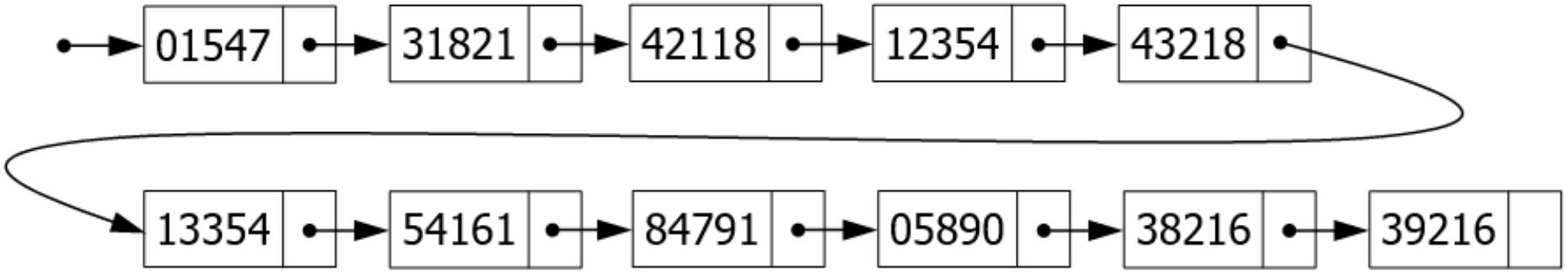


4ª Distribución (2º Dígito):

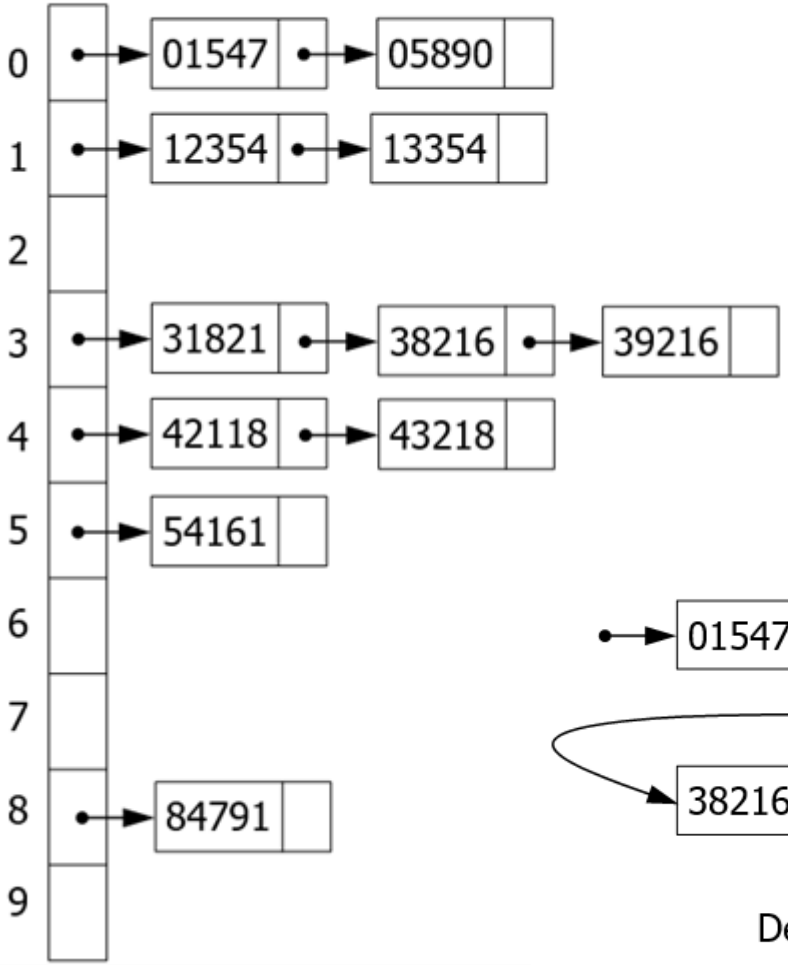


4ª Combinación

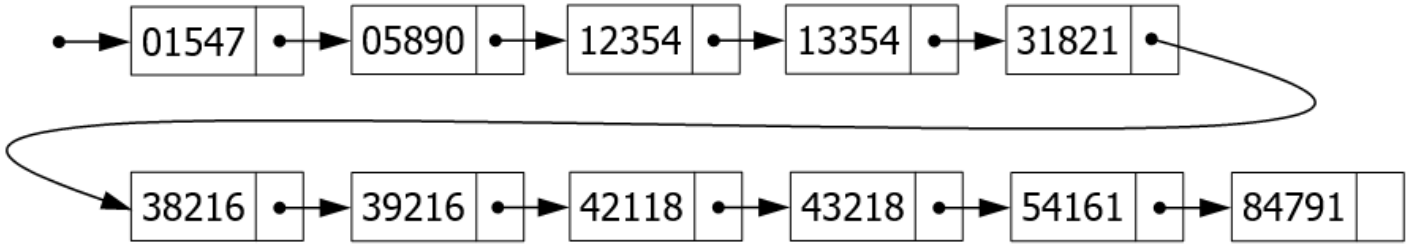




5ª Distribución (1er Dígito):



5ª Combinación



De aquí se pueden escribir los registros al archivo definitivo

5.2.3 RADIX

ACTIVIDADES DE APRENDIZAJE PARA ESCRIBIR EL PROGRAMA COMPLETO

1. Escribir un programa que Genere **n** llaves ***string*** (combinaciones de dígitos del **0 al 9**, cada llave de **6** caracteres de longitud) al azar y las escriba en un archivo en disco llamado **llaves.dat**.

2. Escribir un programa que tome todas las llaves del archivo **llaves.dat** y las guarde en la memoria en una lista encadenada simple. El orden que las llaves traen del archivo se debe conservar, por lo que la lista encadenada debe contar con un apuntador al último nodo para evitar hacer recorridos cada vez que se añade una nueva llave a la lista encadenada. Una vez terminada la carga de las llaves a la lista, hay que imprimir en la pantalla su contenido.

3. **Hacer una distribución en base al último dígito.**
   Tomando como base el programa anterior, escribir uno nuevo que cuente con **10** listas encadenadas (conviene crear dos arreglos de listas con índices del **0** al **9**). Cada llave que se lee del archivo deberá colocarse en una de las 10 listas dependiendo del último carácter de cada llave, es decir, si la llave termina con el dígito **7** se enlazará a la lista de los **sietes**, y así sucesivamente. Finalmente, imprimir el contenido de las 10 listas.

5.2.3 RADIX

4. Modificar el programa anterior para que cuente con una lista nueva a la que llamaremos **lista de combinaciones**, su objetivo es **"conectar"** las listas de los paquetes una vez que se hayan leído todas las llaves del archivo. Al final imprima el contenido de la lista de combinaciones para asegurarse que todo funcione correctamente.
   Observe que el paso de los paquetes a la lista de combinaciones solo implica el cambio de 10 apuntadores, independientemente del número de nodos que haya en las listas.

5. Complete **RADIX** tomando como base el programa anterior (punto 4) para que se repitan las distribuciones y las combinaciones tantas veces como dígitos tenga la llave (considerando cada dígito según corresponda).

Observaciones

- No conviene crear y destruir nodos continuamente mientras se transfieren de una lista a otra, solo hay que cambiar los apuntadores. Hacerlo así redunda en un ahorro muy importante de instrucciones ejecutadas.
- Si la llave está formada por caracteres cualquiera, es importante asegurarse de que todas las llaves sean de la misma longitud.
- El razonamiento, para este método, en cuanto a los espacios a la derecha en llaves tipo cadena es el mismo respecto a los espacios a la izquierda en las llaves numéricas.